CodeWarrior®

Targeting the Java VM From Mac OS

Because of last-minute changes to CodeWarrior, some of the information in this manual may be inaccurate. Please read the Release Notes on the CodeWarrior CD for the latest up-to-date information.

Revised: 98/08/28 dcb

Metrowerks CodeWarrior copyright ©1993–1998 by Metrowerks Inc. and its licensors. All rights reserved.

Documentation stored on the compact disk(s) may be printed by licensee for personal use. Except for the foregoing, no part of this documentation may be reproduced or transmitted in any form by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from Metrowerks Inc.

Metrowerks, the Metrowerks logo, CodeWarrior, and Software at Work are registered trademarks of Metrowerks Inc. PowerPlant and PowerPlant Constructor are trademarks of Metrowerks Inc.

All other trademarks and registered trademarks are the property of their respective owners.

ALL SOFTWARE AND DOCUMENTATION ON THE COMPACT DISK(S) ARE SUBJECT TO THE LICENSE AGREEMENT IN THE CD BOOKLET.

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation 9801 Metric, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	Voice: (800) 377–5416 Fax: (512) 873–4901
World Wide Web	`http://www.metrowerks.com`
Registration information	`register@metrowerks.com`
Technical support	`support@metrowerks.com`
Sales, marketing, & licensing	`sales@metrowerks.com`
CompuServe	go `Metrowerks`

Table of Contents

1 Introduction 7

Read the Release Notes 7
Metrowerks Year 2000 Compliance 7
What's New in This Release 8
What's in This Book. 9
Where to Go from Here 10

2 Getting Started with Codewarrior for Java 13

System Requirements 13
Installing CodeWarrior for Java. 13
Overview of Java in CodeWarrior 14
Development Tools for Java 18
CodeWarrior IDE. 18
Metrowerks Java Compiler 18
Metrowerks JIT Compiler 18
Java Linker 19
Metrowerks Java Application 19
CodeWarrior Debugger 20
Java API Headers. 20

3 Programming Tutorial for Java 21

Applet Description 21
Starting Out 22
Creating a project 23
Changing Target Settings 26
Writing the Applet 30
Adding the Java File 31
Add Graphics Files 34
Editing the HTML File 39
Compile and Run. 42
Oops! What's Happening? 46
Using the CodeWarrior Debugger 47
Exercises 53

4 Creating Java Projects **57**
Types of Java Projects 57
Applets 57
Applications 58
Libraries 58
Using Project Stationery 58
Working with Java in CodeWarrior 59
Creating a New Java Project 60
Creating Java Code 62
Changing Settings 63
Running a Java Project 64
Debugging a Java Project 65
Kinds of Application Projects 66
Using the classes.zip Library 67

5 JavaDoc **71**
CodeWarrior JavaDoc Implementation 71
Using JavaDoc 71

6 Target Settings for Java **77**
Target Settings 78
Java Target 80
Applet 81
Application 82
Library 84
Java Command Line 84
Java Language 85
FTP Post Linker 89
Java MacOS Post Linker 91
JBindery 92
MW Standalone 95
Mac OS Zip 97
Java Output 98
Class Folder 99
Jar File 100
JavaDoc 101

7 Debugging Java Projects 105
Debugger Features and Limitations 105
Special Debugger Features for Java 106
Breaking on Java Exceptions 106
Opening Multiple Class Files in One Browser 107
Choosing a Java Applet Viewer for Debugging 107
Debugging Threads. 108
Viewing Java Virtual Machine Registers 109
Viewing the Java Virtual Machine Disassembly 109
Specifying Java Debugger Settings 111

8 Using Class Wrangler 113
Class Wrangler Window 114
Working with Files and Archives 116
Opening a Zip Archive 117
Creating a Zip Archive 118
Adding Files. 118
Using the Add Files Dialog 119
Add Directory 121
Extracting Files 121
Deleting Files 122
Getting Information on Files 122
Moving Files Between Archives 123
Editing Manifest Files 124
Class Wrangler Preferences 125
Functionality Settings 126
File Filtering Settings 126
Display Settings 128
Miscellaneous Settings 129
Comparing Archives 129

9 Metrowerks Java 131
What Is Metrowerks Java? 131
Utilities for Running Applications and Applets 132
Utilities for Advanced Operations 134
The Java Output window 135

Classpaths Dialog 136
Running Java Applications 137
Using Interpreter to Run an Application 138
Specifying Java Interpreter Options 139
Using the Application Utility 144
Running Applets with Metrowerks Java 144
Running a Local Applet with MW Java 145
Running a Remote Applet with MW Java 145
Changing the Applet's Network Security 146
Disassembling Classes 146
Disassembling a Class 147
Setting the Disassembler Options 148
Verifying Classes 150
Compiling Java Code with the Sun Compiler 151
Compiling a File with the Sun Java Compiler 151
Setting Options for the Sun Java Compiler 152
Generating C Files for Native Methods 154
Metrowerks Java Preferences 156
Program 156
VM 158

10 Troubleshooting on Java 161

Programming Problems 161
Conversion Problems 161
Can't Convert Older Droplet Projects 162

A StandAlone Applets 163

The New JBindery Method 163
About the JBindery Application 163
Creating a Standalone Application 164
The MW StandAlone Method 165
MW StandAlone Features and Limitations 166
Creating a Standalone Application 167

Index 169

1 Introduction

This chapter is your introduction to targeting the Java Virtual Machine (VM) with CodeWarrior. It includes the following topics:

- Read the Release Notes
- Metrowerks Year 2000 Compliance
- What's New in This Release
- What's in This Book
- Where to Go from Here

Read the Release Notes

Before using CodeWarrior Java, read the release notes. They contain important information about any late-breaking changes.

If you are new to Java in CodeWarrior, or to CodeWarrior in general, we strongly recommend that you read this chapter carefully. You'll get a high-level overview of Java in the CodeWarrior environment. This includes a short, hands-on tutorial.

Metrowerks Year 2000 Compliance

The Products provided by Metrowerks under the License agreement process dates only to the extent that the Products use date data provided by the host or target operating system for date representations used in internal processes, such as file modifications. Any Year 2000 Compliance issues resulting from the operation of the Products are therefore necessarily subject to the Year 2000 Compliance of the relevant host or target operating system. Metrowerks directs you to the relevant statements of Microsoft Corporation, Sun Microsystems, Inc., Apple Computer, Inc., and other host or target operating systems relating to the Year 2000 Compliance of their operating sys-

tems. Except as expressly described above, the Products, in themselves, do not process date data and therefore do not implicate Year 2000 Compliance issues.

For additional information, visit: `http://www.metrowerks.com/about/y2k.html`.

What's New in This Release

There are a lot of changes in this latest version of Java for CodeWarrior, to help you program more efficiently:

- The JavaDoc tool is now packaged in a pre linker. This increases performance since the pre linker does not have to parse source files while generating docs.
- You can now debug Java applets and applications running on any of the following virtual machines: Apple MRJ, MW JIT, Sun JDK, Microsoft VM.
- The Java Command Line post linker has been added to allow you to run command line post linker tools when building your Java projects.
- Changing project settings, adding and removing files, and similar operations are now faster when using the Java Linker.
- JavaDoc can now optionally output its docs as a folder hierarchy rather than a flat folder hierarchy with long filenames. See "Output as folder hierarchy" for details.
- JavaDoc can now optionally generate Macintosh-friendly filenames shortened to 31 characters to allow for cross-platform html docs. See "Generate Mac-friendly filenames" for more information.
- The Java MacOS Post Linker panel has been added to allow for removal of Mac-specific functionality from the Java Linker. This new post linker is now responsible for creating JBindery and MW Standalone applications as well as java zip files with Mac resources.
- The JavaDoc plugin now uses the MRJ VM. Therefore, MRJ 2.0 or later must be installed in the system folder in order for this plugin to work.

Many other improvements have been made to the IDE. See the *IDE User Guide* for more information.

What's in This Book

General description of the book's contents by, summarized in a table (Table 1.1) with cross-references by introductory topic.

Table 1.1 Contents of chapters

Introductory Topic	Description
Introduction	Contents of this manual; general description of the CodeWarrior development tools; where to go next
Getting Started with Codewarrior for Java	Installation and setup for Java
Programming Tutorial for Java	Tutorial on Java programming
Creating Java Projects	Creating Java programs
JavaDoc	How to set up and use JavaDoc
Target Settings for Java	Project settings specific to Java programming
Debugging Java Projects	Debugging Java code with the CodeWarrior Debugger
Using Class Wrangler	General purpose ZIP and JAR file management utility
Metrowerks Java	Runtime interpreter for Java bytecodes and additional utilities
Troubleshooting on Java	Troubleshooting information specific to Java

Introductory Topic	Description
StandAlone Applets	Appendix describing how to create double-clickable applets on the Mac OS
Using System.in	Appendix describing how to use `System.in()` on the Mac OS

Where to Go from Here

The information in this manual is, for the most part, specific to either the Java language or the Java virtual machine as a target. This manual does not cover basic features of either the CodeWarrior IDE or the CodeWarrior Debugger. This manual contains Java-specific information.

You do not have to read the chapters in this manual sequentially. Use this manual as a reference to learn about Java in CodeWarrior, or to answer questions you encounter as you develop Java code.

You'll find all the manuals mentioned in this section on the CodeWarrior Reference CD.

For everyone:

- See the *IDE User Guide* and the *Debugger User Guide* for complete information about the CodeWarrior Integrated Development Environment and debugger.

If you're new to CodeWarrior:

- Look for the CodeWarrior tutorials. You'll find them on your CodeWarrior Reference CD.

For general information on Java programming:

This manual does not teach Java syntax, nor does it introduce you to the classes and methods in the Java API—the calls you use to program the Java virtual machine.

- The CodeWarrior Reference CD contains an electronic copy of *Learn Java on the Macintosh* (Addison Wesley) by Barry Boone. *Learn Java on the Macintosh* starts off with some object programming basics, then introduces you to programming in Java. CodeWarrior also comes with some Java-related Apple Guide files.
- *Discover Programming* from Metrowerks can teach you how to program in Java. You can learn more about Discover Programming at:

  `http://www.metrowerks.com/discover/`
- To learn Java, you may consult the *Java Language Tutorial*. This HTML document is available directly from Sun Microsystems at:

  `http://java.sun.com/doc.html`
- To learn more about the Java API, consult the *Java API Documentation*, available as part of the CodeWarrior documentation. This document is in HTML format and comes from Sun Microsystems.

There are also many third party books on Java programming. Check your local bookstore or favorite online bookstore for a wide selection.

2

Getting Started with Codewarrior for Java

This chapter gives you a brief overview of Java in CodeWarrior, installation, system requirements and tool available to you.

This chapter includes the following topics:

- System Requirements
- Installing CodeWarrior for Java
- Overview of Java in CodeWarrior
- Development Tools for Java

System Requirements

If you can run CodeWarrior, you can target the Java virtual machine. CodeWarrior requires a Motorola 68020 (or better) or a PowerPC 601 (or better) processor. CodeWarrior requires 8 megabytes of RAM, System 7.1 or later, and a CD-ROM drive to install the software.

For optimum performance, we *strongly* recommend that you use a Motorola 68030 (or better) or PowerPC microprocessor, with 16 megabytes of RAM.

Installing CodeWarrior for Java

Use the CodeWarrior installer application and follow the instructions in that application. You can examine the various installation

options to see what each installs. The CodeWarrior Heaven and CodeWarrior Java Heaven options install the Java tools.

Select one of the Java-related options to install all the Java tools. We do not recommend that you install the tools manually. The installer ensures that everything is installed in the proper locations.

Overview of Java in CodeWarrior

Java is both a programming language and, in a virtual sense, a computer platform—a target, in CodeWarrior terminology. As such, Java is unique among the languages and targets supported in the CodeWarrior IDE. You can think of Java as not only a language, but as an instruction set for the Java *virtual machine*.

The virtual machine, commonly called a VM, is an abstract computer microprocessor. Like any silicon computer chip, the virtual machine has an instruction set. The instructions for the Java virtual machine are called *bytecodes*.

A bytecode is simply a stream of formatted bytes that has a precisely defined impact on the virtual machine. For example, there are bytecodes that are branch instructions, that push and pop values on a stack, that load or store values, and so forth. Just like traditional assembly language instructions affect a silicon-based processor, the Java bytecodes are a real instruction set for the Java virtual machine.

The reason for the abstract nature of the virtual machine and its bytecode instruction set is to allow you, the programmer, to create platform-independent code. It is the responsibility of the runtime interpreter (or just-in-time compilers that convert bytecodes to native code on the fly) to replicate in a real operating system the effect that a bytecode has on the virtual machine.

The CodeWarrior bytecode interpreter for the Mac OS is built into the Metrowerks Java application.

If you are an experienced code warrior, you'll see that what you do in the Java environment is a little different from what you have done in the past using C, C++, or Pascal.

The development process for Java software in CodeWarrior has the following basic steps.

1. **Write source code.**

   In CodeWarrior, you write Java source code in exactly the same way you do for C, C++, and Pascal. You use the same project manager and the same source code editor. You get all the power and features of CodeWarrior. To target the Java virtual machine, you simply make the appropriate selection in the proper settings panel.

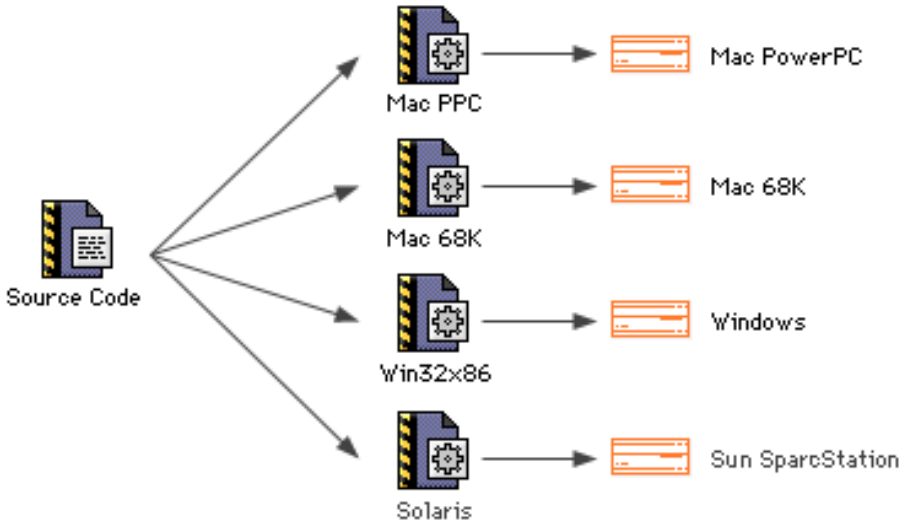
2. **Compile the source code.**

   Again, this process is exactly like it is for any language and target supported in CodeWarrior. You simply issue the appropriate **Compile** or **Make** command from the CodeWarrior Project menu. In response, the CodeWarrior Java compiler generates Java bytecodes based on your source code. CodeWarrior saves the bytecodes in a Java class file.

   The bytecodes in the Java class file are analogous to the object code generated by a C/C++ or Pascal compiler. The difference between traditional object code and Java bytecodes is in how they run.

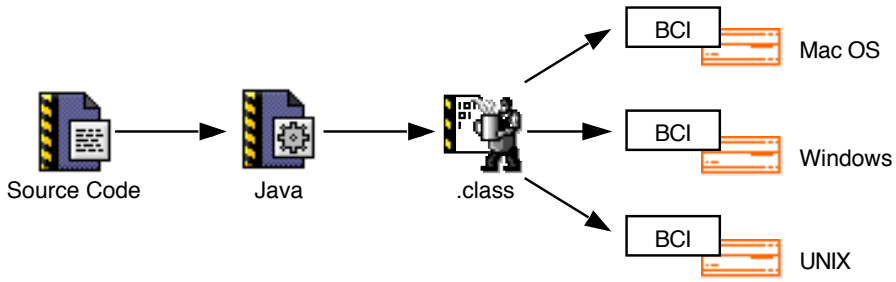
   Object code can run directly on the platform for which it was compiled, but cannot run on any other platform. You must compile it over and over again for each platform. There are also serious cross-platform problems you can encounter trying to address multiple operating systems and computer microprocessors.

Figure 2.1 Compiling and running object code



Java bytecodes cannot run directly on any real machine. On the other hand, the Java bytecodes created by the CodeWarrior Java compiler can be interpreted to run on *any* platform that has a byte-code interpreter (BCI). You write the code once, compile it once, and then you're done.

Figure 2.2 Compiling and running Java bytecodes



3. **Run the code.**

   Just choose **Run** from the Project menu and CodeWarrior launches the applet with the applet viewer you've chosen. To learn how to specify a the applet viewer, see "Target Settings."

   Remember, the bytecodes in the class file cannot run directly on any real platform. The class file must be run under the control of a bytecode interpreter (BCI). The BCI is responsible for translating the bytecodes into machine-native instructions. The CodeWarrior bytecode interpreter is contained in the Metrowerks Java application. Metrowerks Java is a separate application that CodeWarrior can launch for you.

4. **Debug the code with the CodeWarrior Debugger.**

   The final step in the development process is to test and debug your code. You debug Java code using the same CodeWarrior debugger you use for C/C++ and Pascal code. The CodeWarrior debugger understands Java. The debugger does everything you would expect the debugger to do. It has a stack crawl, displays local variables, and allows you to set breakpoints.

   To debug Java code, your applet or application must be built to include debugging information. Choose **Enable Debugger** from the Project menu to tell the Java compiler that you want to create debugging information when it compiles your code. Then recompile your project. You can choose **Make** from the Project menu to compile and link your code. You may also choose **Debug** from the Project menu, and CodeWarrior compiles, links, and launches your program under debugger control.

   From here you can set break points, and run your program.

Summary

That's all there is to it. As you can see, the big difference between Java and compiled languages is that Java code is interpreted at runtime.

If you have used CodeWarrior in the past, developing Java code in CodeWarrior is going to look very familiar to you. If you are new to CodeWarrior, you'll find developing Java code to be a pleasure once you master the tools.

Development Tools for Java

These are the tools CodeWarrior provides for developing Java software:

- CodeWarrior IDE
- Metrowerks Java Compiler
- Metrowerks JIT Compiler
- Java Linker
- Metrowerks Java Application
- CodeWarrior Debugger
- Java API Headers

CodeWarrior IDE

The CodeWarrior Integrated Development Environment (IDE) provides a complete set of tools for developing application programs for a variety of target platforms, including 68K- and PowerPC-based Mac OS systems, Win32/x86 systems, Solaris, and Java. You use the same IDE when developing code for all target platforms; you designate the platform and code generator of your choice when you create your project.

For information on the CodeWarrior IDE, see the *IDE User Guide*.

Metrowerks Java Compiler

The Metrowerks Java compiler is a plug-in tool built exclusively for use in the CodeWarrior IDE.

Metrowerks JIT Compiler

The Metrowerks Just-In-Time compiler (JIT), included as part of the Java Compiler, seeks to accelerate Java execution by translating Java bytecodes into native machine code. It does this on an incremental, method by method basis. The first time a method is invoked, the JIT compiler is run instead, transforming it into native code. Once transformed, this native code runs in place of the bytecodes. For all

future invocations during the lifetime of the Java VM session, the native code is run instead.

The JIT compiler does some simple optimizations in order to improve the quality of the code it produces. One of the most significant features is, mapping the Java stack to registers. With this feature, it is often possible to map several Java VM instructions to one native machine instruction. The JIT compiler knows about a number of frequently occurring patterns which it rewrites into sequences to produce better code.

NOTE: There are some commonly used Java VM instructions that are large, resulting in a 12:1 code expansion over bytecodes.

The JIT compiler also performs Common Subexpression Elimination (CSE) for arrays. This means the JIT does not reevaluate duplicate array indices.

For example, if you have two or more occurrences of `a[i]` where `i=2`, the JIT evaluates the first occurrence of `a[2]` then uses that value in place of all other occurrences of `a[2]`.

Java Linker

Like the compilers, the Java Linker is a plug-in tool integrated into the CodeWarrior IDE.

Metrowerks Java Application

Metrowerks Java is a many-faceted application that contains a tremendous amount of functionality.

Most importantly, the Metrowerks Java application is the runtime interpreter for Java bytecodes. You use this application to open and run Java applets and applications. The Metrowerks Java application incorporates an applet viewer, among other features. In many cases, all you will use this aplication for is to run java applets and applications.

However, the Metrowerks Java application has many other features. For example, it includes a Java bytecode disassembler, Sun Java Compiler, and more.

The Metrowerks Java application is located in `Metrowerks CodeWarrior:Metrowerks Java:Metrowerks VM`. For full information, see "Metrowerks Java."

CodeWarrior Debugger

The CodeWarrior debugger controls your program's execution and allows you to see what's happening internally as your program runs. You use the debugger to find problems in your program's execution.

The debugger can execute your program one statement at a time, and suspend execution when control reaches a specified point. When the debugger stops a program, you can view the chain of function calls, examine and change the values of variables, and inspect the contents of the processor's registers.

For general information about the debugger, including all of its common features and its visual interface, you should read the *Debugger User Guide*.

For more information about debugging software for Java, see "Debugging Java Projects."

Java API Headers

The Java Application Programming Interface (API) is a set of data structures and functions used to interface to the Java operating system.

3

Programming Tutorial for Java

This tutorial takes you very quickly through the CodeWarrior Java environment. This tutorial does not teach you Java programming. It is designed to teach you how to use the CodeWarrior Integrated Development Environment (IDE) to write and debug Java code. This tutorial takes you step-by-step through the entire process. The applet you create is designed to be immediately reusable with little to no change, depending on your needs.

NOTE: This tutorial requires a Java enabled browser or Java applet viewer. If you do not already have a Java enabled browser installed, the latest version of Microsoft Internet Explorer is available on the CodeWarrior CD. We recommend you install it before continuing with this tutorial.

The topics discussed in this tutorial are:

- Applet Description
- Starting Out
- Writing the Applet
- Compile and Run
- Oops! What's Happening?
- Exercises

Applet Description

The applet you will create in this tutorial is a simple navigation menu, called an Image Map, that you can add to any web page. It consists of two graphics, one for the menu in its unhighlighted state,

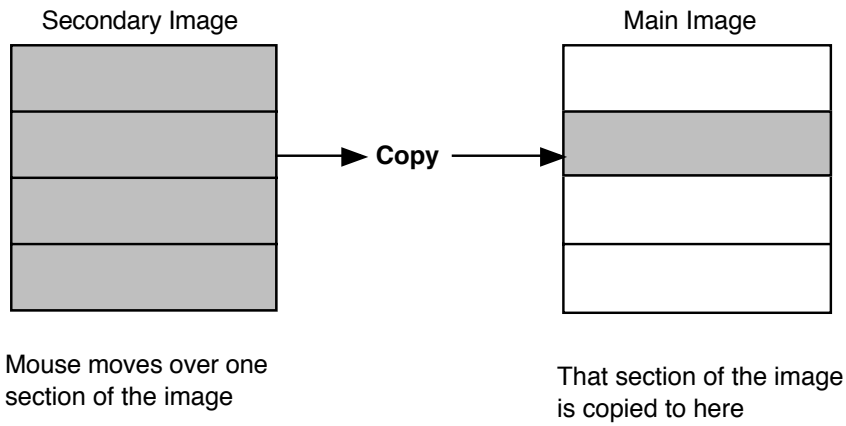
and one which will be used to draw each menu item in its highlighted state. The original idea for this tutorial comes from a similar applet written by Chris Cason on the Pov-Ray home page located at:

`http://www.povray.org`.

Theory of Operation

How the applet works is very simple. It loads two graphics into memory. The first graphic is an image of the menu, listing all choices. The second graphic is the same image with some minor differences (in this case color) to show the current menu choice. As the mouse passes over each option in the menu, the applet copies the corresponding area from the secondary graphic and "paints" it over top of the original. This provides more visual feedback when moving the mouse over a link area which is superior to guessing where to click and hoping you're right. Figure 3.1 shows how the applet works.

Figure 3.1 Theory of operation



Starting Out

Before you begin, locate the ImageMap Tutorial folder. It is in the following location: `CodeWarrior Examples:CodeWarrior Java:Metrowerks Sample Projects:Java Tutorial:`Im-

ageMap Tutorial. If this folder is not already installed on your hard drive, copy it from the CodeWarrior Reference CD to your hard drive. The files you create and modify during this tutorial will be located in this folder.

All of the files for the completed tutorial can be found in the `ImageMap Solution` folder. You can compare your work against those files if you run into problems.

WARNING! You should be very careful to use the exact names specified in each step of this tutorial. Some features of Java are very dependent on names, and the applet may not work if you change names. Java is also a case-sensitive language, so make sure you match case as well as spelling.

Let's get started. The topics covered in this section of the tutorial are:

- Creating a project
- Changing Target Settings

Creating a project

Locate the CodeWarrior IDE application and launch it. After it opens, choose **New Project** from the File menu. The New Project dialog box appears (Figure 3.2).

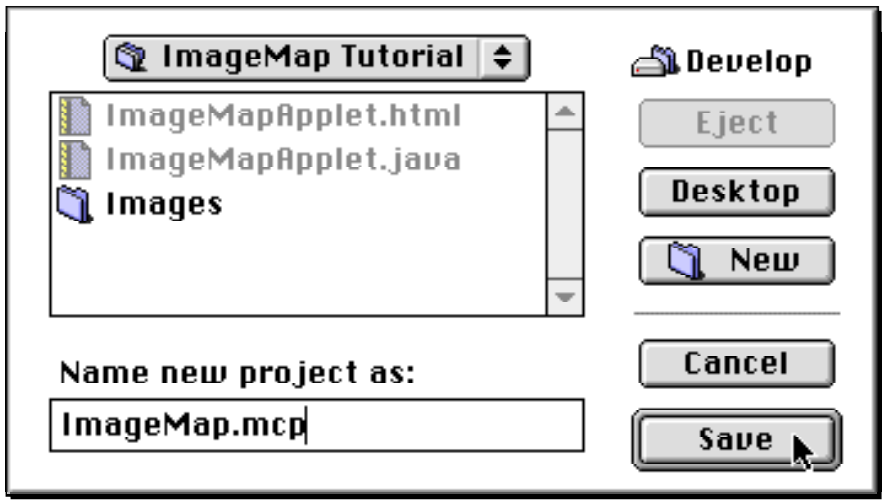
Figure 3.2 The New Project dialog box

Select the Java Applet stationery. If the stationery isn't visible, click the disclosure triangle beside the Java group to make it visible, then select it. Also make sure the **Create Folder** check box is *off*. Then click **OK**.

CodeWarrior now displays a save file dialog. Navigate to the `ImageMap Tutorial` folder on your hard drive. The dialog should look like the one in Figure 3.3. Name your project `ImageMap.mcp`.

NOTE: The "`.mcp`" suffix is a CodeWarrior naming convention for project files. If you do not add "`.mcp`", CodeWarrior will add it for you.

Figure 3.3 The name new project dialog box



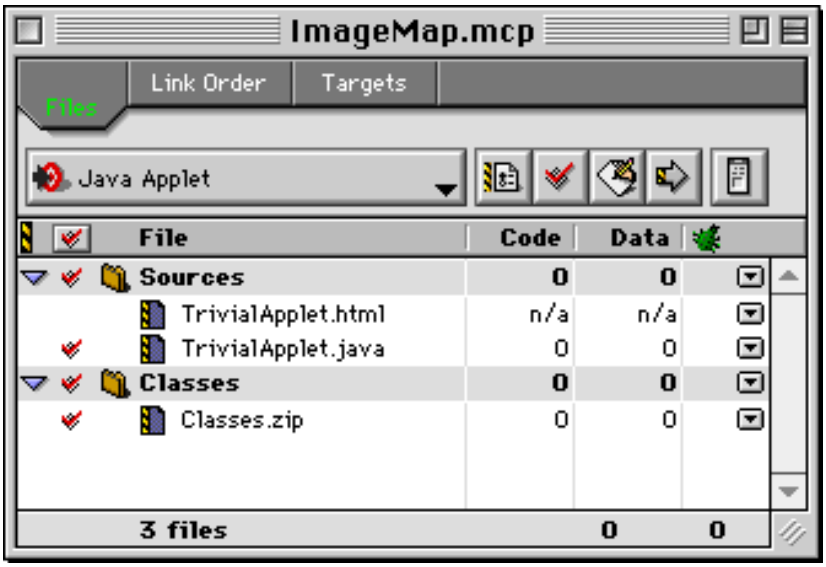
Click the **Save** button. CodeWarrior creates three files in the tutorial folder:

- The project file `ImageMap.mcp`
- The HTML file `TrivialApplet.html`
- The Java file `TrivialApplet.java`

CodeWarrior also creates a "data" folder. This folder contains files with information about your project file, target settings you've chosen, object code, and browser information. You should not change the contents of this folder.

Code Warrior opens the project and displays a window similar to that shown in Figure 3.4. You've got a project under way!

Figure 3.4 The project window



Changing Target Settings

A project file can have multiple build targets. For example, you can have a *debug* build target and a *release* build target in the same project. Each build target has its own unique collection of settings that control how CodeWarrior creates the final executable. These are the build target settings. When you change the settings for a build target, the change affects only the current build target.

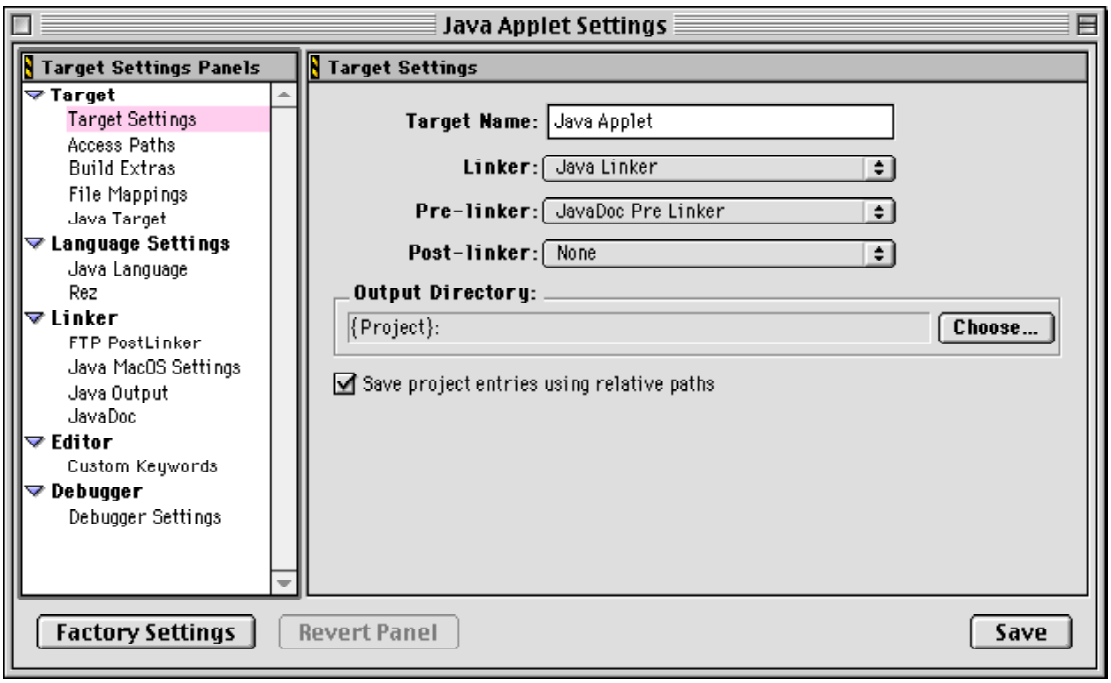
NOTE: CodeWarrior also has global preferences that affect all projects. For more information on preferences and target settings, see the IDE User Guide.

By creating a new project using stationery, the default build target settings are already set for you. However, for most projects, you will need to change a few of these settings in order to get the results you are looking for.

Choose ***MyTarget*** **Settings** from the Edit menu, where ***MyTarget*** is the name of the currently active build target in the front-most

project. In this case, the build target name should be **Java Applet** . CodeWarrior displays the Target Settings dialog box (Figure 3.5).

Figure 3.5 Target settings dialog box



The Target Settings dialog box contains a list of *settings panels* on the left. The right side of the dialog box shows the options for the currently selected panel.

For the purposes of this tutorial, we're only interested in changing a few options in three of these panels. For a more detailed description of each settings panel, see "Target Settings for Java."

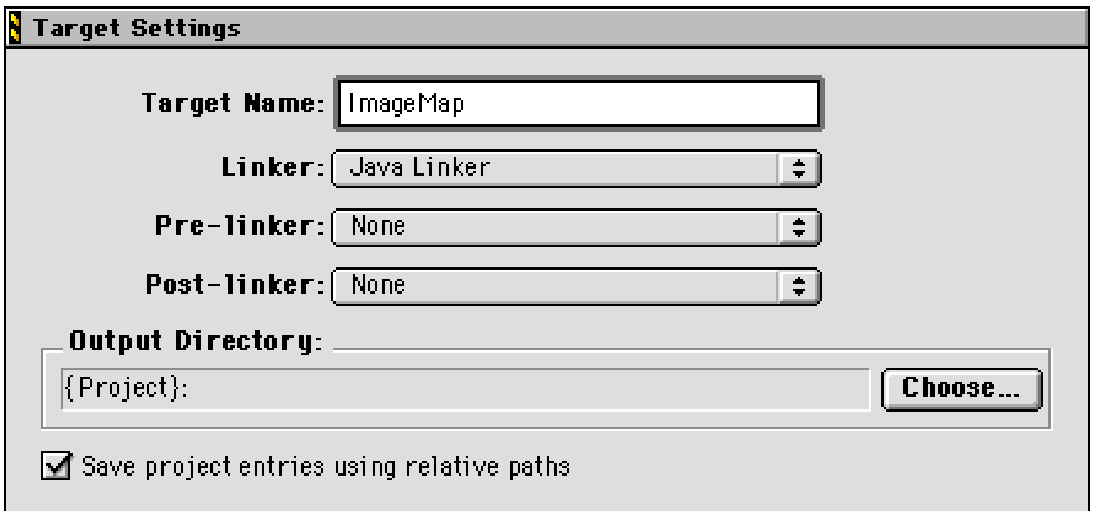
Changing target information

Here, you change three settings for this target: the name, the applet viewer, and the output. These settings are in three different panels.

1. **Change the target name.**

   Click **Target Settings** from the list on the left. Change the **Target Name** field to "`ImageMap`" (Figure 3.6).

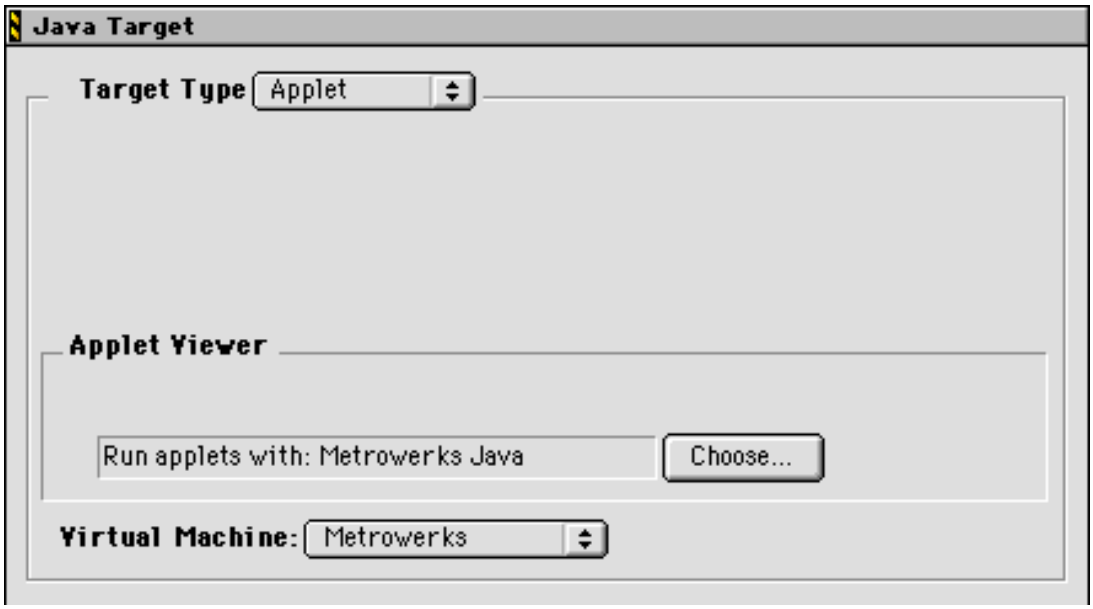
Figure 3.6 Target Settings panel



2. **Change applet viewer.**

   Click **Java Target** from the list on the left. Set **Applet Viewer** to be the Metrowerks Java application by clicking the **Choose** button, finding the Metrowerks Java application and clicking **OK**. The panel should look like Figure 3.7.

Figure 3.7 Java Target panel

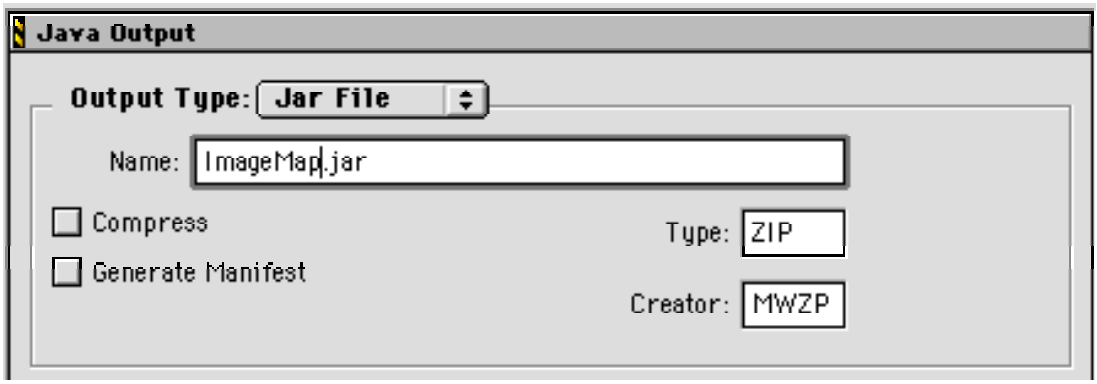


3. **Change the output.**

   The only other settings panel we are interested in is the **Java Output** panel in the Linker section. Select the **Output** panel in the list of settings panels.

   The **Output** panel controls the type and name of the file CodeWarrior creates when it links your code. Make sure **Type** is set to **Jar File** and the **Name** field is set to `ImageMap.jar` (Figure 3.8). These settings create a new Java Jar archive in same folder as your project.

Figure 3.8 Output panel



Now that all the settings have been changed, you're ready to start writing the code. Click the **Save** button at the bottom of the Target Settings dialog box. Close the Target Settings dialog box.

NOTE: Depending on which options were changed, you may see the following dialog upon closing the dialog. Clicking the Cancel button will cancel the closing of the dialog and allow you to make more changes. Clicking the OK button will close the dialog.

Figure 3.9 Relink applet dialog

Target "Java Applet" must be relinked.
Changes to the settings of target "Java Applet" require that it must be linked again. Do you want to continue?
Cancel
OK

Writing the Applet

In this section, you write the actual code for the applet. Topics discussed in this section of the tutorial are:

- Adding the Java File
- Editing the HTML File

Adding the Java File

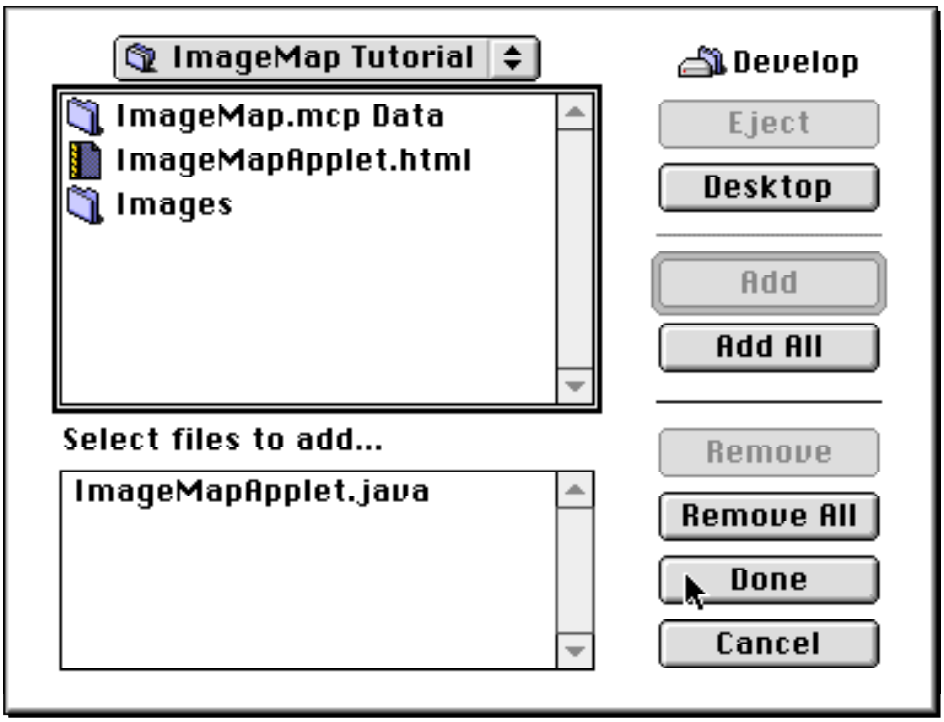
The code for the ImageMap applet is extensive at almost 200 lines. To make things easier, the code is provided for you in the tutorial folder. You add this file to the project.

1. **Add ImageMapApplet.java to the project.**

   To make sure your file is added to the appropriate section of the project window (in this case, in the Sources folder), first highlight any file in that section or the section title itself. Then choose **Add Files** from the Project menu. The Add Files dialog box shown in Figure 3.10 will appear.

   Navigate to the `ImageMap Tutorial` folder. Select `ImageMapApplet.java` and click the **Add** button. Then click the **Done** button.

Figure 3.10 Add files dialog box



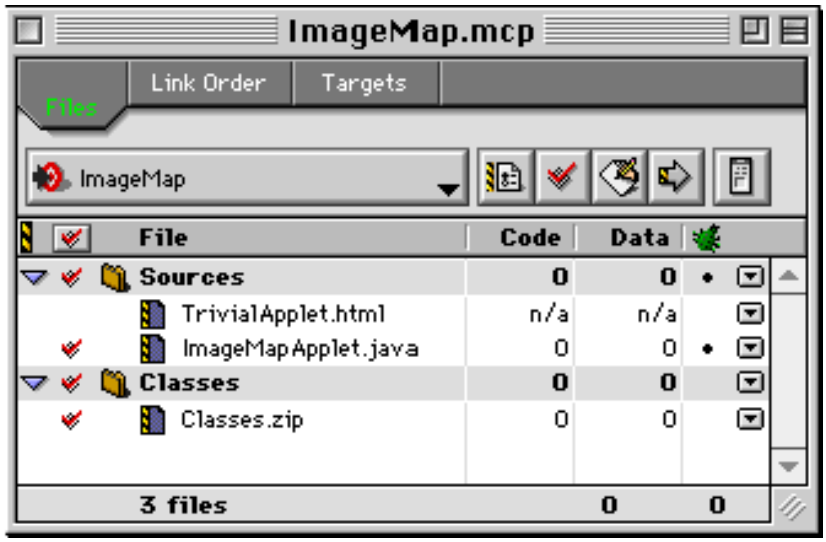
2. Remove placeholder file.

Now, remove the `TrivialApplet.java` file from the project by selecting it, and then choosing **Remove Selected Items** from the Project menu. The resulting project window should look like Figure 3.11.

TIP: For a faster way to remove items in the project window, select the file or item you wish to remove. Hold down the Option key and press delete.

The file `TrivialApplet.java` is still in the `ImageMap Tutorial` folder. You can delete it if you want, but leaving it there will not affect your project.

Figure 3.11 Java code added to the project



Examining the code

Open the `ImageMapApplet.java` file and look at the source code.

NOTE: Do not make any changes to the source code yet.

Reading the comments in the code will help you understand the functionality of the applet. If you aren't familiar with Java syntax, you may not understand all the code. That's OK. The purpose of this tutorial is not to teach you Java, but to introduce you to CodeWarrior. See "Where to Go from Here" for references to Java language documentation.

The applet uses the Java API to display a picture. When you move the cursor over certain defined areas of the picture, that area changes. The images used can be found in the `Images` folder.

Java naming conventions

The name of your Java file is important. The name of the file must be the same as the name of the class created in the file, and end with

the `.java` extension. This extension is a Java programming convention for naming source files. The Metrowerks Java compiler uses this extension to recognize Java source files more easily.

NOTE: Actually, the Metrowerks Java compiler lets you name a file whatever you want. This feature is useful on the Mac OS, which limits file names to 32 characters: too short for some Java class names. However, you should name a Java file after its class whenever possible, so your code is compatible with all Java compilers. If you must give a file a different name, choose a name that's as close as possible to the class's name.

You have all the code for a complete, interactive Java applet. You're most of the way there. There are just two more things to do before you test the applet.

Add Graphics Files

Part of the flexibility in using a Jar file for the applet is that it can contain files that are not Java Classes. For this project, we want to include the two graphic files used for the image map in the archive.

There are two ways to accomplish this. You can add the graphics manually by editing the Jar file with a ZIP utility, or you can add them directly to the CodeWarrior project file to be merged into the final jar file when you build the project.

In order to add these graphics directly to the CodeWarrior project, there are two things that need to be done. First, the current target in the project must be able to recognize the graphic file. Second, after the files are added, you need to tell CodeWarrior to merge the graphics into the Jar file.

Recognizing graphics files

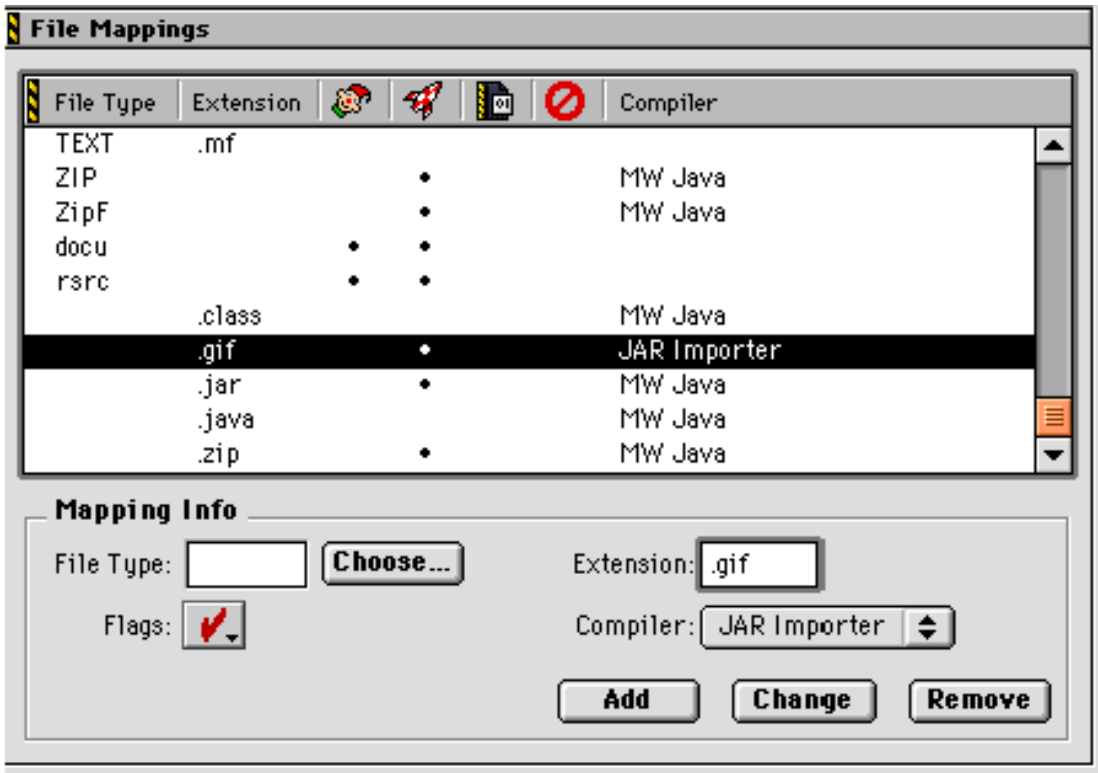
The **File Mappings** settings panel allows you to add, change, or remove file types recognized by your build target by using a character extension added to the file name, such as `.gif`, `.jpg`, or `.html`. The default file mappings are set for every new target in a

CodeWarrior project file. You can change these settings on a per target basis.

Choose ***ImageMap*** **Settings** from the Edit menu (remember, ***ImageMap*** is the name of the currently active target in your project). Choose File Mappings from the list of settings panels on the left. Look over the list of file mappings. Make sure there is an entry for `.gif` as shown in Figure 3.12. If there isn't an entry, type in the data as shown. Make sure the Launchable flag is the only one with a check mark in the **Flags** popup menu. When you have entered all of the necessary data, click the **Add** button.

TIP: Setting the Launchable flag allows CodeWarrior to launch your favorite application that handles that file type when you double-click on it in the Project window.

Figure 3.12 File Mappings panel



Setting up the graphic files

Now that the file mappings are set to recognize the two graphic files, they can be added to the project. However, it is important to add the graphics to a new group that mirrors their physical directory structure.

1. **Create new group**

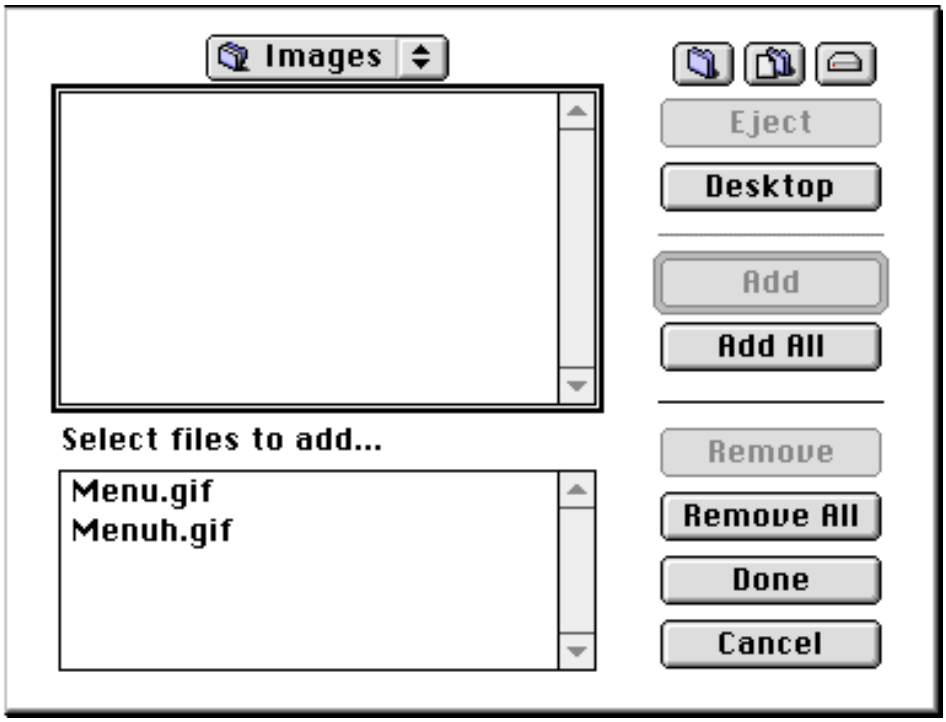
   Choose **Create New Group** from the Project menu. Name the group "Images" as shown in Figure 3.13. Click **OK**.

Figure 3.13 New group dialog

2. **Add menu graphics to the new group.**

   Click on the Images group in the Project window to select it. Choose **Add Files** from the Project menu and add both graphic files (Figure 3.14). The files will be added to the Images group.

Figure 3.14 Adding graphics



3. **Set merge flag**

   To select both graphics files, hold the Shift key down and click once on each graphics file in the Project Window. Choose Project Inspector from the Window menu. Click the Merge Into Output checkbox and click **OK** (Figure 3.15). Close the window.

Figure 3.15 Merge setting

Now, when the applet is compiled, the graphic images will be added directly into the Jar file under the heading "Images."

Editing the HTML File

In this section, you edit `TrivialApplet.html` and rename it. The HTML file is used by the web browser to pass parameters to your Java applet.

CodeWarrior allows you to add non-code files to a project. In this way you can keep track of all files related to a project, even those that are not compiled. The project stationery for an applet project includes a sample file, `TrivialApplet.html`.

When developing an applet, you typically want to have a small HTML file that embeds the applet code. CodeWarrior uses this file to run your program from within the IDE. When you choose **Run** from the Project menu, CodeWarrior opens the first HTML file in the **Link Order** view of your project. The order of HTML files in the **Files** view of your project does not matter.

For more information on either the File view or Link Order view of the CodeWarrior project window, see the *IDE User Guide*.

1. **Edit TrivialApplet.html**

   Double-click `TrivialApplet.html` in the project window to open it. `TrivialApplet.html` is a simple HTML file that displays an applet and gives you a link to its source code. Delete the code in `TrivialApplet.html` and replace it with code shown in Listing 3.1.

TIP: If you don't feel like typing in the code, you add it to the project using the **Add Files** command. The `ImageMap.html` file is supplied in the `ImageMap Tutorial` folder for this purpose.

Listing 3.1 The HTML file for ImageMapApplet

```
<html>
<head>
<title>Java Tutorial</title>
</head>
<body bgcolor="#CC9966" text="#000000" link="#0000FF"
vlink="#000080" alink="#FF0000">
<basefont size=2>
<div align=left>
<applet archive="ImageMap.jar" code="ImageMapApplet.class"
width=248 height=172>
<param name="normal" value="images/menu.gif">
```

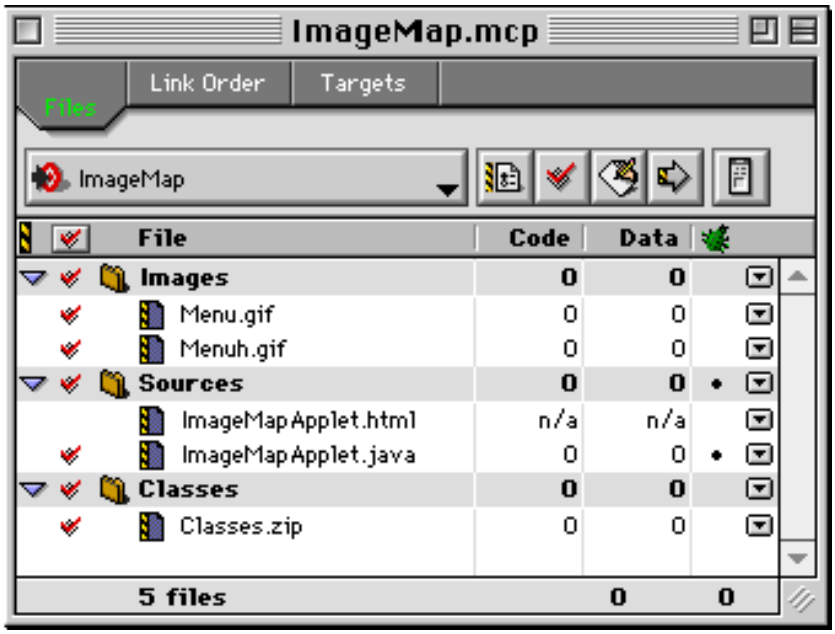
```
<param name="highlighted" value="images/menuh.gif">
<param name="URL0" value="http://www.metrowerks.com/">
<param name="URL1" value="http://www.gamelan.com/">
<param name="URL2" value="http://search.yahoo.com/bin/
search?p=java">
<param name="URL3" value="http://java.sun.com/docs/index.html">
</applet>
</body>
</html>
```

The HTML code tells the Web browser that the applet code can be found in a file named `ImageMap.jar`. It also sets the viewing size of the applet to be 278 by 172 pixels.

2. **Save the file under a new name**

   Choose **Save As** from the File menu. Make sure you're in the `ImageMap Tutorial` folder. Name the file `ImageMapApplet.html`. CodeWarrior saves the file with the new name and changes the name of the file in the project window, as shown in Figure 3.16.

Figure 3.16 The project window with the renamed HTML file.



You've removed the `TrivialApplet.html` file from your project; but it is still in the `ImageMap Tutorial` folder. You can delete it if you want, but leaving it there doesn't affect your project.

Compile and Run

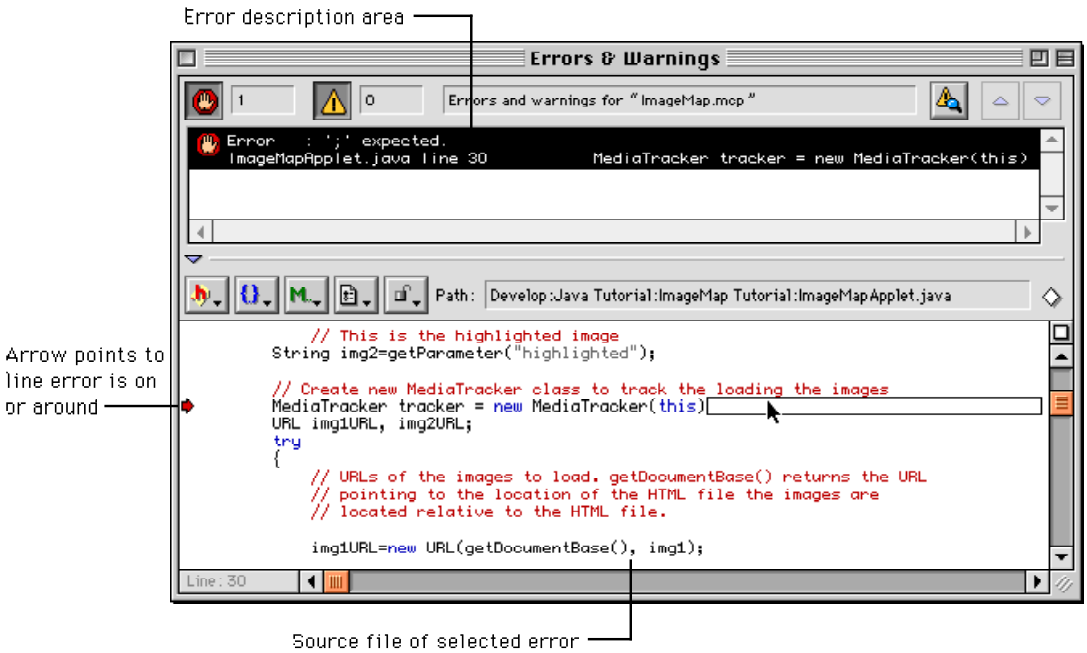
You are now near the final stages of your first Java applet, and this tutorial. All that is left to do is compile and test it to make sure it runs smoothly.

Choose **Make** from the Project menu. The **Make** command compiles and links your code into the finished applet, all in one step.

NOTE: The CodeWarrior IDE is very smart about picking up syntax errors at compile time. Usually these errors are the result of mistyped or unused variable names, incorrect class information, or

other related errors. The tutorial code has one deliberate error (Figure 3.17).

Figure 3.17 Compiler errors


Error description area
Errors & Warnings
1
0
Errors and warnings for "ImageMap.mcp"
Error : ';' expected.
ImageMapApplet.java line 30
MediaTracker tracker = new MediaTracker(this)
Path: Develop:Java Tutorial:ImageMap Tutorial:ImageMapApplet.java
// This is the highlighted image
String img2=getParameter("highlighted");
// Create new MediaTracker class to track the loading the images
MediaTracker tracker = new MediaTracker(this)
URL img1URL, img2URL;
try
{
// URLs of the images to load. getDocumentBase() returns the URL
// pointing to the location of the HTML file the images are
// located relative to the HTML file.
img1URL=new URL(getDocumentBase(), img1);
Line: 30
Arrow points to line error is on or around
Source file of selected error

The Errors & Warnings window displays a list of errors the compiler found when it tried to make the project, in this case only one. The bottom part of this window shows the source file that contains the selected error. The red arrow points to the specific line of code where the error was encountered. Sometimes the error is on the same line, but it can also be found a line or two before depending on the type of error.

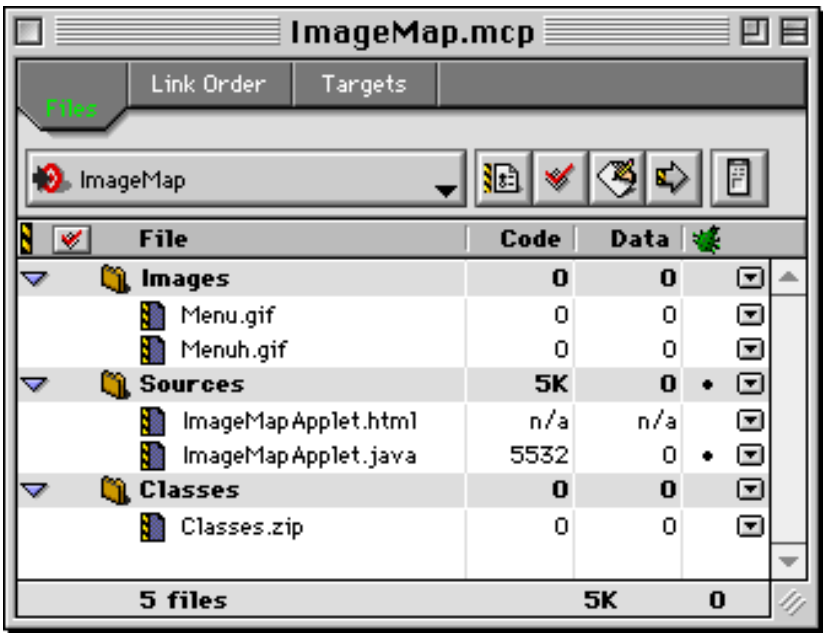
The source view of the Errors & Warnings window is completely editable. You can edit and save your code directly from this window. There is no need to open up the source file and try to find the error manually. This is a great time saver if you have many errors.

In this case, the error is a missing semicolon (;) at the end of the line pointed out by the red arrow. Click at the end of the line and type a semicolon (;). Then choose **Save** from the File menu to save the file. Close the Errors & Warnings window, then choose **Make** from the Project menu to compile the project again.

While building your project, the check mark to the left of the files in the project window disappears, indicating that the files no longer need to be compiled. In the Code column, the number of bytes of code is updated. The number is zero for the `classes.zip` file because that is a shared library to which the applet links at runtime. This file contains the Java classes and methods called from the `ImageMapApplet.java` source file.

Your project is completely compiled and ready to run (Figure 3.18).

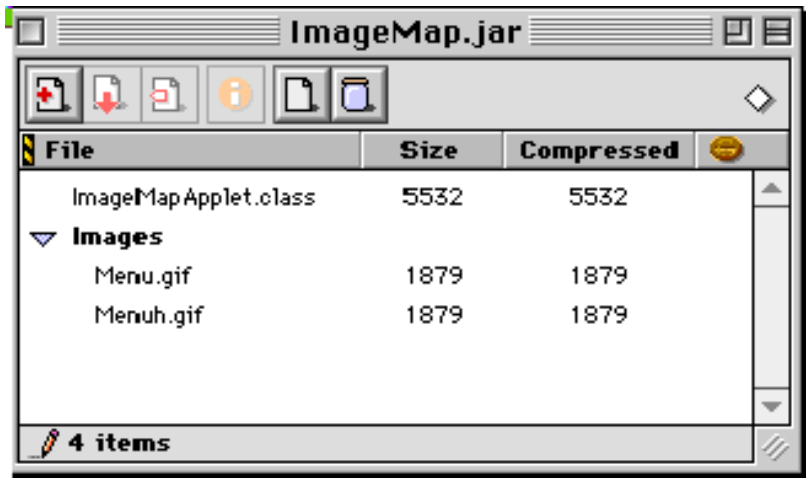
Figure 3.18 Compiled java project



Examining the output

Take a minute to examine the Jar file produced by CodeWarrior with your favorite ZIP tool. It should look similar to Figure 3.19. Notice that the graphic files have been merged into the file in a directory structure matching the name of the group they were under in the project window. Notice also that the Jar file is not compressed. Do not compress the file! While you save disk space using a compressed Jar, the applet will run very slowly.

Figure 3.19 Jar file



Run the applet

The last thing to do at this point is test whether the applet works. Choose **Run** from the Project menu. This launches the Metrowerks Java application using the HTML file in your project. The HTML file, in turn, tells the Metrowerks Java application where to find the class file and graphics used in the applet.

NOTE: If you are connected to the Internet via modem, either with SLIP or PPP, your applet viewer may try to start a connection. You do not have to be connected to the Internet for this part of the tutorial, though you can be if you wish. If you don't want to be con-

nected, just cancel the connection. The applet will continue to work without it.

Do not make any selections in the applet at this time. Move the cursor over each word in the graphic. The word changes color from blue to yellow, indicating the current selection. However, there is a bug in the applet.

Oops! What's Happening?

As you pass the cursor over "Gamelan," the word changes to show a half highlighted state and the text is displaced (Figure 3.20). The bug that's causing this problem could be in one of several places in the code. Now, you have to debug the applet to find out what is going on.

Figure 3.20 Applet bug

There is one primary topic discussed in this section:

- Using the CodeWarrior Debugger

Using the CodeWarrior Debugger

The CodeWarrior Debugger is integrated into CodeWarrior to help you debug your code. The same debugger works for Java, C, C++, Pascal, and assembly language.

The debugger is called a "source level debugger" because it displays your source code so you can see exactly which line will be executed next. This has the added benefit of getting a clearer understanding of your code, without the need to learn assembly language.

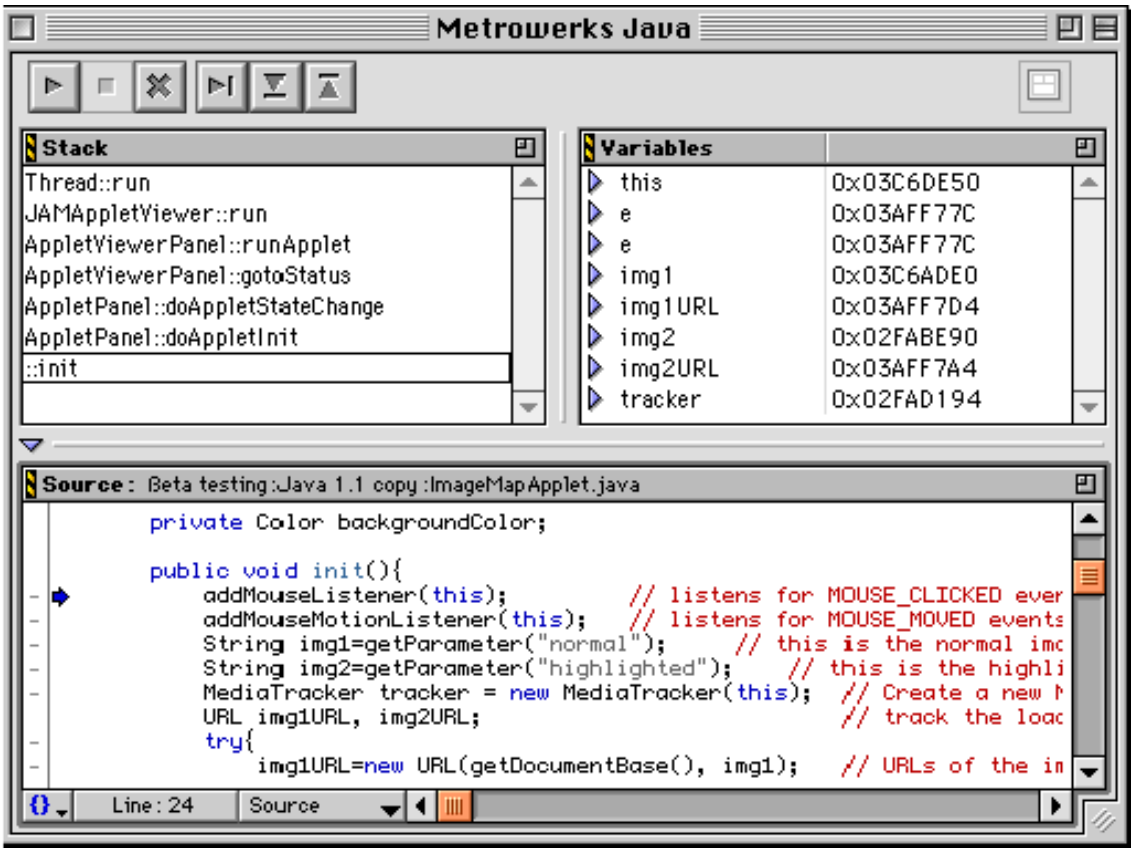
For more information on the debugger, see the *Debugger User Guide*.

Debugging the applet

Before you can debug your Java code, you must first tell CodeWarrior to use the debugger when running the applet. Choose **Enable Debugger** from the Project menu. Now you are ready to start a debugging session. Choose **Debug** from the Project menu. CodeWarrior relinks the project and opens the debugger.

CodeWarrior opens the debugger and passes the first HTML file in the Link Order view of your project to the debugger. The debugger launches the Metrowerks Java application which loads the applet and creates a window for it to run in. Finally, the debugger comes back to the front, displaying the program window for the applet, with the current statement arrow at the first line of code in the `init()` method, as shown in Figure 3.21.

Figure 3.21 The debugger's program window



The debugger allows you to examine local variables, step through the program, set breakpoints, and perform other debugger functions. The top left pane is the Stack Crawl Pane. The Stack Crawl pane shows the order in which each method has been executed, called a *call chain*. The arrow in the bottom pane is called the *current statement arrow*. This arrow points to the line of code that will be executed next. See the *Debugger User Guide* for a more detailed description of the debugger windows.

Solving the Problem

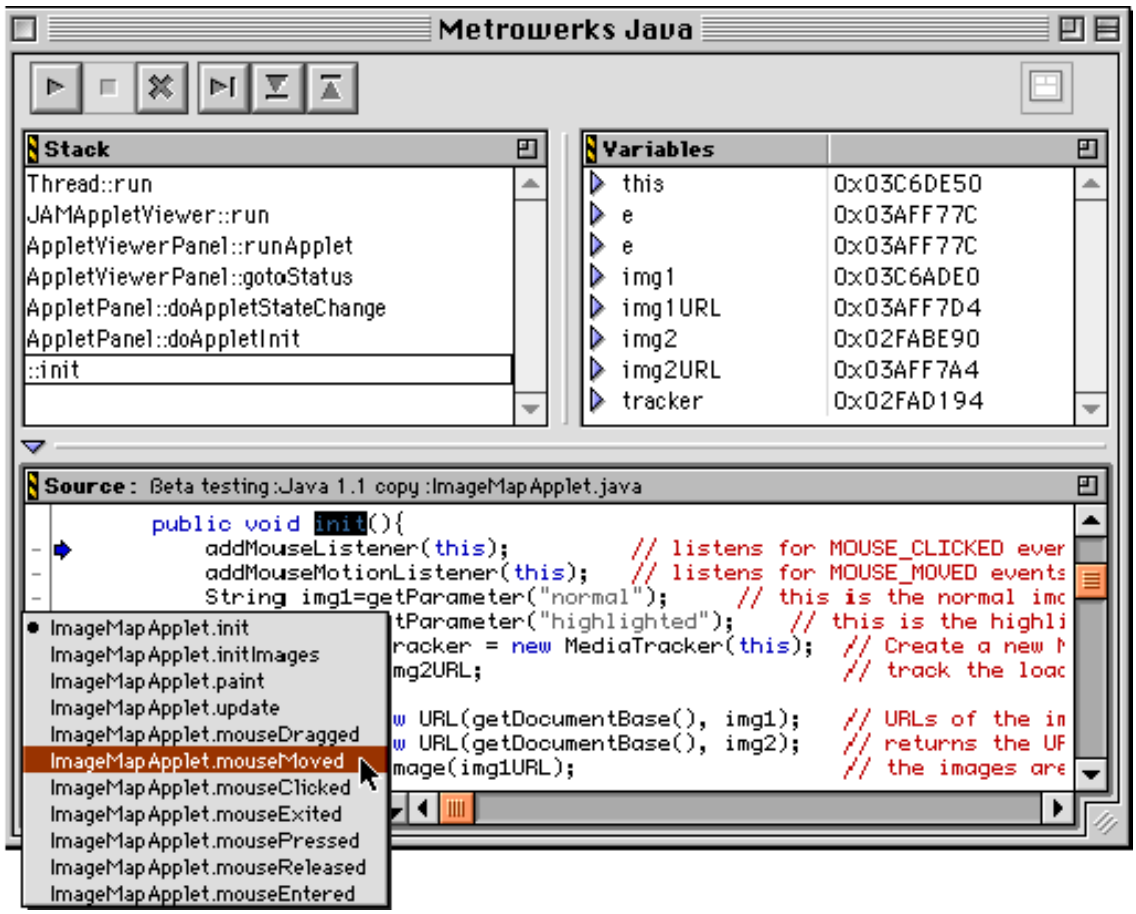
To solve the bug in this applet, it's necessary to review what happened and when. This gives you greater understanding of the code and a better idea where to start looking for the problem.

Remember that the applet essentially seems to run correctly with the exception for one item. As the mouse moved over the menu graphic, it changed to display an incorrect image. That's where to start looking, the `mouseMove()` method.

1. **Select the `mouseMove()` method in the debugger.**

   The bottom left corner of the debugger window contains a small icon. This icon is a pop-up menu. Click on the pop-up menu and choose the `mouseMove()` method (Figure 3.22). The debugger scrolls the source code to show the first few lines of the selected method.

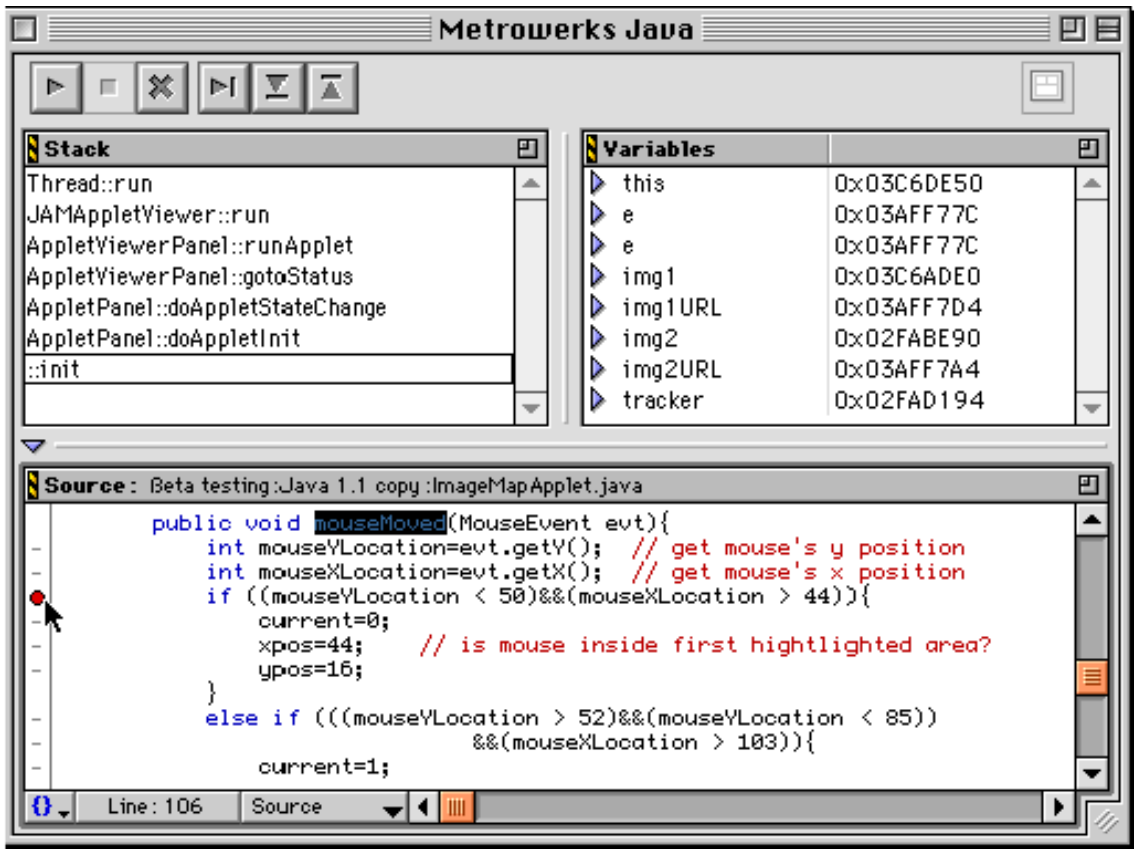
Figure 3.22 Choosing a method



A *breakpoint* is like a stop sign. It tells the debugger to execute all code up to the line just before the breakpoint. Each dash (-) symbol in the left column indicates a line of code where a breakpoint can be set.

You want to set a breakpoint at the first available line of code in the `mouseMove()` method. To set the breakpoint, simply click the dash symbol, as shown in Figure 3.23.

Figure 3.23 Setting the breakpoint

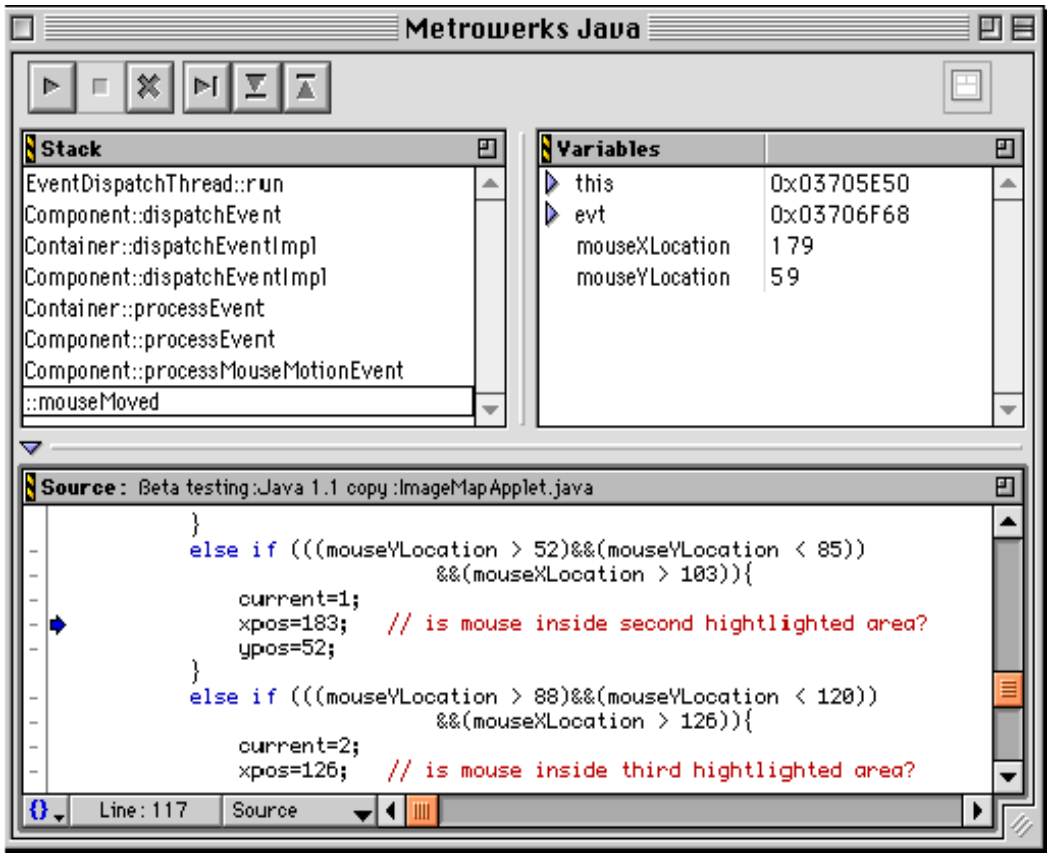


Choose **Run** from the Project menu. This allows the Metrowerks Java application to run the applet until it reaches the line of code where you set the breakpoint. It won't reach that line of code until you move the cursor over the applet.

Move the cursor over the second item in the list of the graphic. The debugger becomes the front-most application and the current statement arrow is on the same line as the breakpoint. At this point, you want to examine each line of code as it executes to try to find the error. This process is called *stepping* through your code.

Choose **Step** from the Control menu. The debugger executes that line of code. Continue to use the **Step** command until you reach the line of code shown in Figure 3.24.

Figure 3.24 Offending line of code



This is the culprit! This line tells the applet to act as if the cursor is at a position of 183 pixels on the X axis. Really, this should be 103! Small errors like this are common when typing code and can cause unnecessary headaches in your own applications.

Editing the source file

The debugger is not an editor. You cannot edit your code directly in the debugger window. Halt execution of the applet by choosing Kill from the Debug menu. Then open the file by double-clicking it in the project window.

Change the line of code as shown in Listing 3.2.

Listing 3.2 Corrected java code

```
xpos = 103;
```

Recompile and run

Now that you've changed some code, it's necessary to recompile the applet and run it again. This time, you run it without the debugger.

The Metrowerks Java application is not a web browser, thus clicking on menu selections will not bring you to the corresponding sites. Set Applet Viewer in the Java Target settings panel to your favorite Java enabled web browser.

NOTE: If you do not already have a Javaenabled browser installed, the latest version of Microsoft Internet Explorer is available on the CodeWarrior CD.

Choose **Disable Debugger** from the Project menu. Choose **Run** from the Project menu. If you have an internet connection, let your web browser start the connection. Now move the cursor over the graphical menu. Try clicking on each item in the menu to see where it leads. **Congratulations** ! You've completed your first Java Applet!

Exercises

There is still more that can be done with this applet. Currently, clicking on a navigational link brings up a second browser window. It would be better if the navigation menu was always at the left of the window and each site appeared in a frame at the right. Create a new HTML file that sets up the web page using Frames. There is an example of this in the `ImageMap Solution` folder called `Applet-Frames.html`.

Double Buffering

Rather than draw directly to screen, the applet code uses a technique known as double buffering to minimize the flickering. The

applet draws to an off-screen location (in memory) and then, when the image is fully drawn, draws the whole image on the screen at once.

Change the code so that it does not use double buffering to see what happens. To make this change, remove the code indicated by the comments in Listing 3.3.

Listing 3.3 Screen flicker

```
public class ImageMapApplet extends Applet
{
  private Image image, hImage // remove this ->, offscreenImage;
//...

public void init()
  {
//....

// remove the following line
offscreenImage=createImage(imgWidth, imgHeight);
```

New parameters

The way the applet is currently set up, you need to recompile it every time you change the graphic, whether you add items, delete items, use different sizes, etc. The code that sets this up is `initImages()`. It would be nice if you didn't have to do this. It would make the applet more usable and portable to just change the parameters in the HTML file and leave the source code alone (Listing 3.4).

Listing 3.4 HTML area parameters

```
<PARAM name="sections" value="4"> // the number of highlighted
sections in the image
<PARAM name="coords0" value="0, 0, 428, 60">// the coordinates for
each
<PARAM name="coords1" value="0, 60, 428, 60">
```

```
<PARAM name="coords2" value="0, 120, 428, 60">
<PARAM name="coords3" value="0, 180, 428, 60">
```

Then you would need a method to parse these coordinates (Listing 3.5).

Listing 3.5 Value parser shell code

```
String temp;
temp=getParameter("sections");
int s=Integer.parseInt(temp);
for (int i=0; i<s; i++){
  temp=getParameter("coords"+i);
  // parse string for numbers here
  // this can be done using methods from the String and Integer
classes
  Do_Clip_Image(x, y, w, h);
}
```

4

Creating Java Projects

This chapter explains how to create and manage any Java project, and how to set the options in the Java Project settings panels.

This chapter contains these sections:

- Types of Java Projects describes the types of Java projects you can create with CodeWarrior.
- Using Project Stationery describes each project stationery option in the New Projects dialog box.
- Working with Java in CodeWarrior describes how to create and manage any Java project from start to finish.
- Using the classes.zip Library describes what's in classes.zip, the library file that's in most every Java project, and how to add it in a project.

Types of Java Projects

This section discusses the various types of code objects that you can create for Java. They include the following:

- Applets
- Applications
- Libraries

Applets

An Applet is a small Java application that runs over the Web, usually through a Web browser. Applets can consist of multiple class files or a single Jar file.

Applications

A Java Application are executables that usually run through a Java Interpreter. Applications can consist of multiple class files or a single Jar file.

Libraries

A Java Library is a collection of Java classes that can be used in many projects. Libraries are usually compiled into zip files (like `classes.zip`) or Jar files.

Using Project Stationery

The **New Project** dialog box (Figure 4.1) displays a list of several different pre-configured projects and their associated files, called *Project Stationery*.

Figure 4.1 New Project dialog box

Each set of project stationery is pre-configured for different types of Java projects. This saves you time when creating your own project files as you only need to change a few options instead of setting up a project from scratch.

Table Table 4.1 lists each project stationery option.

Table 4.1 Project stationery options

Stationery	Description
Java Applet	Pre-configured settings for a Java applet. Output is a Jar file.
Java Application	Pre-configured settings for a Java application. Output is a Jar file.
Java App-Applet	A multi-target project (Applet & Application) that demonstrates how to use `AppletFrame.java` to turn your applet into an application.
Java Library	Pre-configured settings for a Java library. Output is a Jar file.

Working with Java in CodeWarrior

This section describes how to create, run, and debug any kind of Java project, including applets, applications, and libraries. The topics in this section are:

- Creating a New Java Project
- Creating Java Code
- Changing Settings
- Running a Java Project
- Debugging a Java Project
- Kinds of Application Projects

Creating a New Java Project

The following procedure describes how to create a new Java project. The project can be either an applet, an application, or a library of Java code.

1. **Create a new project.**

   Choose **New Project** from the File menu. CodeWarrior displays the New Project dialog.

2. **Select project stationery.**

   In the New Project dialog box, select the proper stationery for the kind of project you want to create: applet, application, or library. To learn more about various kinds of application projects, see "Kinds of Application Projects."

   You may also decide whether to have CodeWarrior create a new folder for you in which to place all your files. Typically you want to create a new folder.

   When you click the **OK** button, CodeWarrior displays a file dialog.

3. **Name the project.**

   In the file naming dialog, enter a name for the project, and click **OK**.

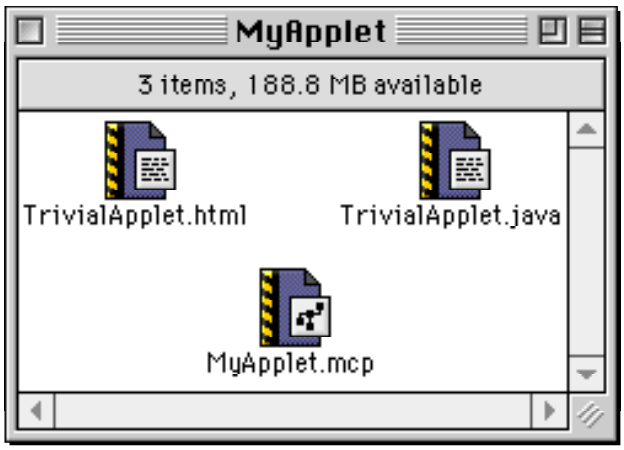
TIP: If you want to compile your project with all versions of CodeWarrior, the project suffix must be `.mcp`.

If you had the **Create Folder** check box on in the New Project dialog box, CodeWarrior creates a new folder. Inside that folder, CodeWarrior puts a new project file (with all project settings identical to the stationery project you selected), and duplicates any files included in that stationery project.

For example, for an applet, you get a new project file, an HTML file, named `TrivialApplet.html`, and a Java source file named `TrivialApplet.java`. Figure 4.2 shows you the result when you use the Java applet stationery to create a project named MyApplet.

The project file has the name you entered in the dialog box. The new folder has the same name as the project file without any naming extension.

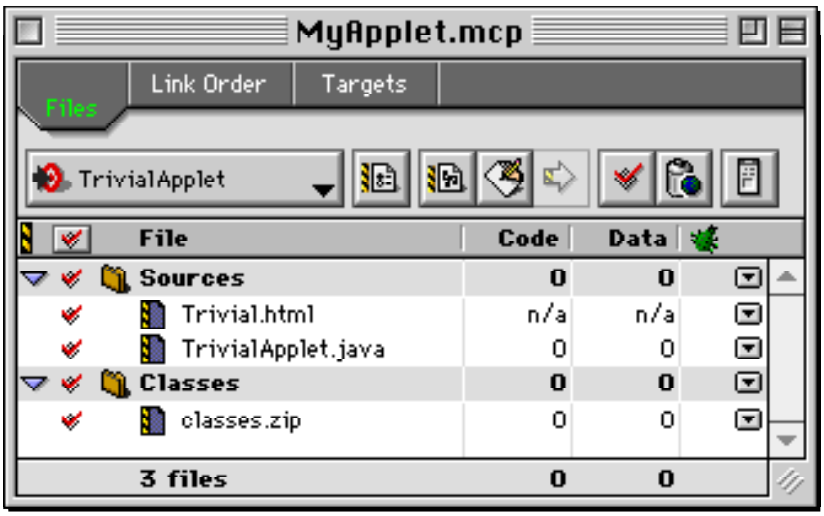
Figure 4.2 The Java Applet folder



WARNING! The project manager also creates a directory named MyProject , where “MyProject” is the name of your project. CodeWarrior uses this directory to store internal data. You should not modify or move any file in the data directory.

After creating the files and folder, CodeWarrior displays a new project window, like the one below:

Figure 4.3 The Java Applet project



The stationery projects hold example files. In typical use you modify these files and save them with a new name, or replace them with your own files.

The `classes.zip` file is a shared library of standard Java classes your applet can use. For more information, see "Using the classes.zip Library."

Creating Java Code

To create Java code, use the CodeWarrior tools: the editor, the browser, and the project manager. For details on these tools, see the *IDE User Guide*.

In typical practice, you start with a project based on stationery, which includes example or "placeholder" files. To learn how to create a project in this way, see "Creating a New Java Project."

For example, if you are creating an applet based on the Java applet stationery, there are two such files: `TrivialApplet.java` and `TrivialApplet.html`.

The `TrivialApplet.java` file contains a sample Java applet. You can open the file, replace its contents with your own code, and save

the file under a new name. You can add any other source file as well.

NOTE: Each Java file must contain only one public class. The file name must be the same name as the public class with the suffix `.java`. For example, if the name of your class is MyClass, the Java file must be named `MyClass.java`. If you don't follow this convention, CodeWarrior may compile your project anyway, but other Java compilers may not.

Similarly, the `TrivialApplet.html` file contains sample HTML code. You can modify this file so that it displays your applet. You typically specify both the location of the applet (a folder name) and the name of the class containing the main function. You can set the name of your classes folder by opening the **Target Settings** dialog box, and modifying the name specified in the **Output** panel. See "Target Settings" and "Java Output" for more information.

Changing Settings

If you use project stationery as a starting point, you generally won't need to change many of the settings. The default settings are perfect for most projects. However, this is not the case for all settings or all projects. This is especially true if you have more than one target in a project. You can modify a variety of settings for Java projects. Each panel is explained in detail in "Target Settings for Java."

Common settings you might change are the main class (for Java applications) and the output name (the name of your final program). For applications you may also choose whether to output your files in a class folder, or in a zip file.

Specifying the main class

To tell CodeWarrior the name of your main class for a Java application, go to the **Java Target** panel, make sure that you are creating an application, and set the name in the **Main Class** field in the panel. For more information, see "Main Class."

Specifying output name

To tell CodeWarrior what to name your final output file (be it an applet or an application), go to the **Java Output** panel and enter the name in the **Name** field. For more information, see "Java Output."

Specifying output type

To tell CodeWarrior whether to create a class folder or a Jar file, go to the **Java Target** panel and set the Type option. For more information, see "Java Target."

Running a Java Project

You may run applets and applications. You cannot run libraries. To run a project, choose **Run** from the Project menu. At that time, CodeWarrior compiles and links any changed files, and launches your program. Precisely how your program launches depends upon whether it is an applet or an application.

Running a Java applet

When developing an applet, you typically have a small HTML file that embeds the applet code. CodeWarrior uses this file to run your program from within the IDE. When you choose **Run** from the Project menu, CodeWarrior opens the first HTML file in the Link Order view with an HTML browser or applet viewer. In other words, the order of HTML files in the File view in the Project Window does not matter. It is the order as shown in the Link Order view in the Project Window.

The default applet viewer is the Metrowerks Java application. To learn how to modify this setting, see "Applet."

NOTE: For applet projects, if there is no HTML file in your project, the **Run** command is disabled.

You can add as many HTML files as you want to a project. If your applet is complex, you may want several HTML files that test different aspects of your applet. To change the HTML file that CodeWar-

rior uses, simply rearrange the files in the Link Order view of your project window.

NOTE: You can also run an applet by opening one of its HTML files with any Java-enabled browser.

Running a Java application

To run an application, choose **Run** from the Project menu in the CodeWarrior IDE or double-click it in the Finder. The Metrowerks Java application loads the application's code and runs the `main` method of the application's main class. If you entered a value in the **Main Class** field in the Java Project settings panel, it will use that class. Otherwise, it will assume the main class is the first class in the first file in the project window.

If you run a droplet by dragging and dropping files onto it, it also passes the full pathnames of those files as arguments to the main class's `main` method. Note that if you run a droplet by double-clicking it or choosing **Run** from the IDE's Project menu, you cannot specify arguments for it.

NOTE: To run an application, the Metrowerks Java application must be installed on the computer.

To run a zip file or class folder, you must use the `javai` window in the Metrowerks Java application, described in "Running Java Applications."

Debugging a Java Project

You debug an applet project the same way you debug any other project. For more information on the CodeWarrior debugger, consult the *Debugger User Guide*. For more information on Java-specific features in the debugger, see "Debugging Java Projects."

To debug a Java project, follow these steps:

1. **Turn on debugging.**

   This step can be skipped if debugging is already enabled.

   Choose **Enable Debugger** from the Project menu. You may see a dialog box telling you that certain settings must be modified for debugging. Click OK, and CodeWarrior sets up debugging automatically. In the project window, black dots appear in the debugging column next to the source files.

2. **Run the project under the debugger.**

   Choose **Debug** from the Project menu. CodeWarrior compiles any changed files and generates debugging information for all files that have a black dot in the debugging column of the project window. CodeWarrior launches your project using the appropriate mechanism for your project. For a discussion of this process, see "Running a Java Project."

   Once the project is running, control returns to the debugger. You can now debug the applet like you would any other program. For more information on the debugger, see the *Debugger User Guide*. For more information on Java-specific features in the debugger, see "Debugging Java Projects."

NOTE: If you want to debug a droplet that's been launched by drag and drop, see "Debugging a Running Java Program" on page 101.

Kinds of Application Projects

CodeWarrior lets you create Java applications that you can run without an HTML browser. Generally, you'll create a *droplet*, a small application that automatically launches the Metrowerks Java application to run its code. If you drag and drop files onto a droplet, it will pass the file names as arguments to the application's `main` method.

These are the other types of applications you might create for specific purposes:

- A *Jar file* is an archive of all your classes. It is a convenient way to transfer and organize all the class files and any associated other files in one package.
- A *class folder* is a folder that contains all your classes. You cannot run it by double-clicking it. You must use the javai window in the Metrowerks Java application, described in "Running Java Applications." Choose it if you need to run the file under other computer platforms that can't use zip files.

All these files types contain Java bytecodes that will run on any computer platform that supports Java. However, application files also contain Mac OS resources that tell the Metrowerks Java application which class contains the `main()` method. A zip file or a class folder doesn't contain these resources and you must somehow tell the interpreter which class contains the `main()` method. Use an application when you will use the file only on a Mac OS computer. Use a zip file or class folder when you'll use the file on other computer platforms as well.

Using the classes.zip Library

The `classes.zip` file is a library of Java classes that all Java programs share. It includes all the packages in the Java API, such as the Abstract Windowing Toolkit (AWT), I/O, and Applet packages.

The location depends on which Java VM you have installed. It is usually in a Classes folder in a subfolder of your Extensions folder.

Almost every Java program that you create uses `classes.zip`. However, `classes.zip` is a large file, about 1.5 MB, so you probably don't want to copy it into every one of your Java applications and applets.

To instruct CodeWarrior not to merge the `Classes.zip` file into the final output, first select the `Classes.zip` file in the project window. Then choose **Project Inspector** from the Window menu to open the **Project Inspector**. Make sure that the **Merge Into Output** checkbox is not selected. CodeWarrior won't copy the library into your finished program.

To enable your project to compile without the `Classes.zip` library, make sure that the **Import Weak** checkbox is selected. With these settings, your program will automatically look for the library on the host system when it is executed.

The **Project Inspector** window should now appear as in Figure 4.4.

Figure 4.4 The Import Weak Classes Setting

When you create a project with project stationery, `classes.zip` is automatically set up this way for you.

NOTE: The **Initialize Before** option is ignored by the CodeWarrior Java compiler. **Merge Into Output** will add `Classes.zip` to the final binary.

5

JavaDoc

JavaDoc is a batch compiler written in Java by JavaSoft™. It processes Java source code files, and uses the comments preceding classes, methods, etc. to generate HTML based documentation for the code. The Java API documentation on your CodeWarrior CD was generated by this compiler.

There is one primary topic discussed in this section:

- CodeWarrior JavaDoc Implementation
- Using JavaDoc

CodeWarrior JavaDoc Implementation

CodeWarrior implements JavaDoc as a Pre-Linker because of the batch nature of the JavaSoft compiler. The JavaSoft version of JavaDoc must know about all the compiled files simultaneously. In the current CodeWarrior plugin architecture, a compiler works on one file at a time, but a linker knows about all compiled files. So JavaDoc in CodeWarrior is implemented as a pre-linker.

Another added benefit of implementing JavaDoc as a pre-linker is speed. Creating JavaDoc files should be the last thing you do in with your Java project. Assuming this is true, your code will have already been compiled and tested. At this point, when you implement JavaDoc, the process is very fast as everything the pre-linker needs to know was already done at compile time. No need to recompile, just relink.

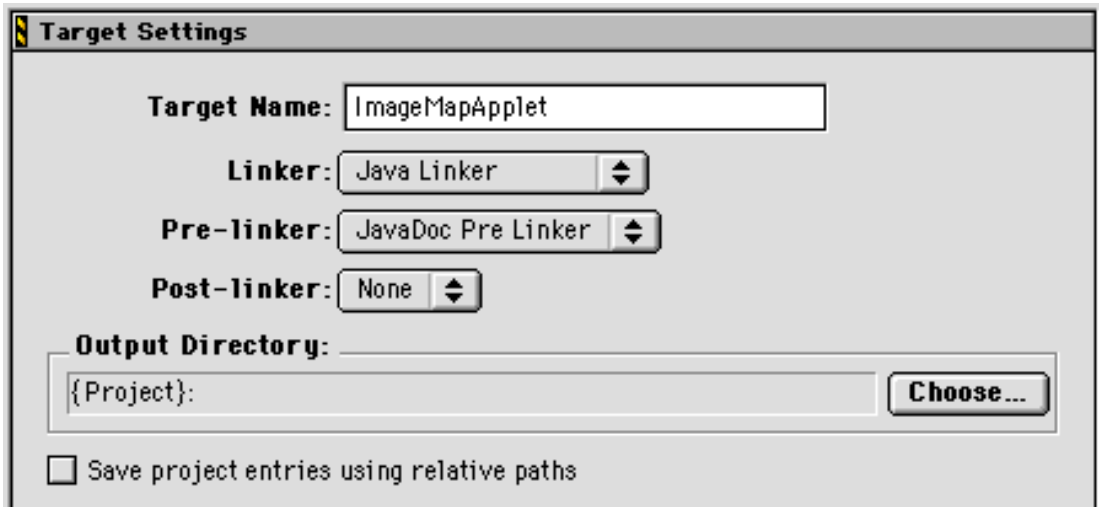
Using JavaDoc

Using JavaDoc is as simple as choosing the JavaDoc pre-linker, and compiling the files.

1. **Choose pre-linker for JavaDoc target.**

   Choose ***Target*** **Settings** from the Edit menu, where ***Target*** is the name of your current Java target. Choose **JavaDoc Pre Linker** from the **Pre-Linker** popup menu in the Target Settings panel (Figure 5.1). Click Save to save changes. Close the Target Settings dialog box. Click **OK** for subsequent warning dialogs.

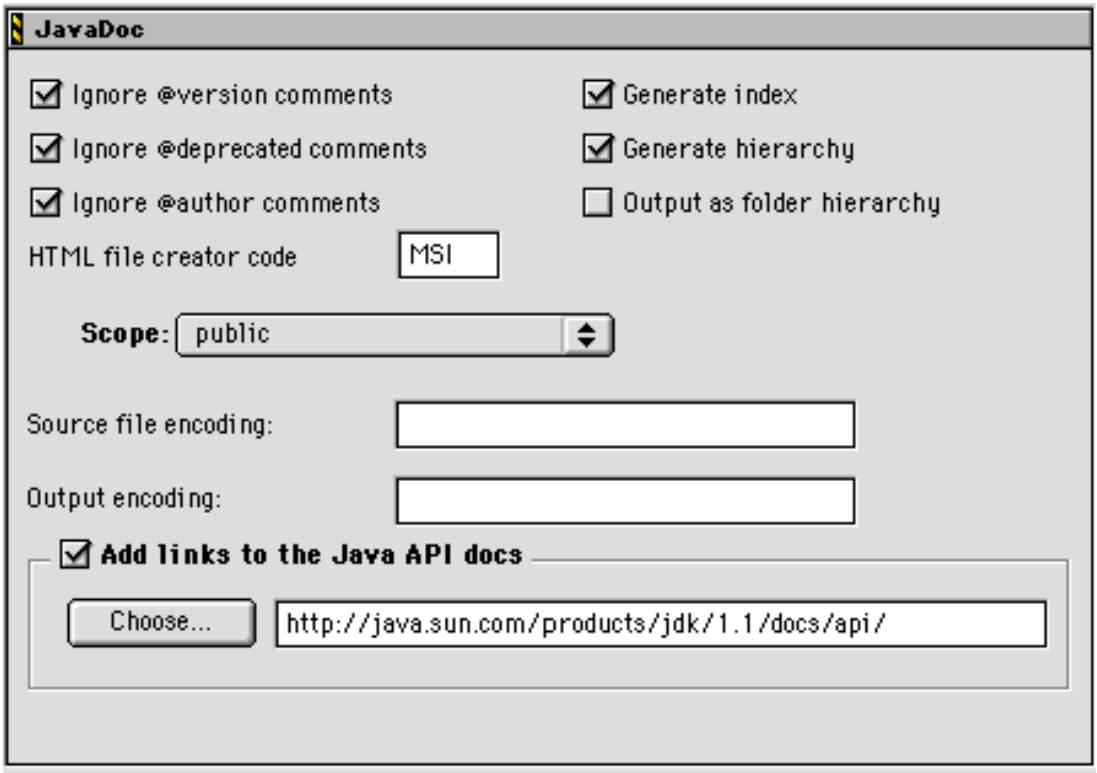
Figure 5.1 Choosing JavaDoc linker



2. **Set JavaDoc Project options.**

   Choose the settings you wish to use for JavaDoc from the **JavaDoc** panel (Figure 5.2). For a full description of each setting in this panel, see "JavaDoc."

Figure 5.2 JavaDoc Project panel



Choose **Make** from the Project menu. JavaDoc creates a new folder in the same folder as your project called "Docs" (Figure 5.3).

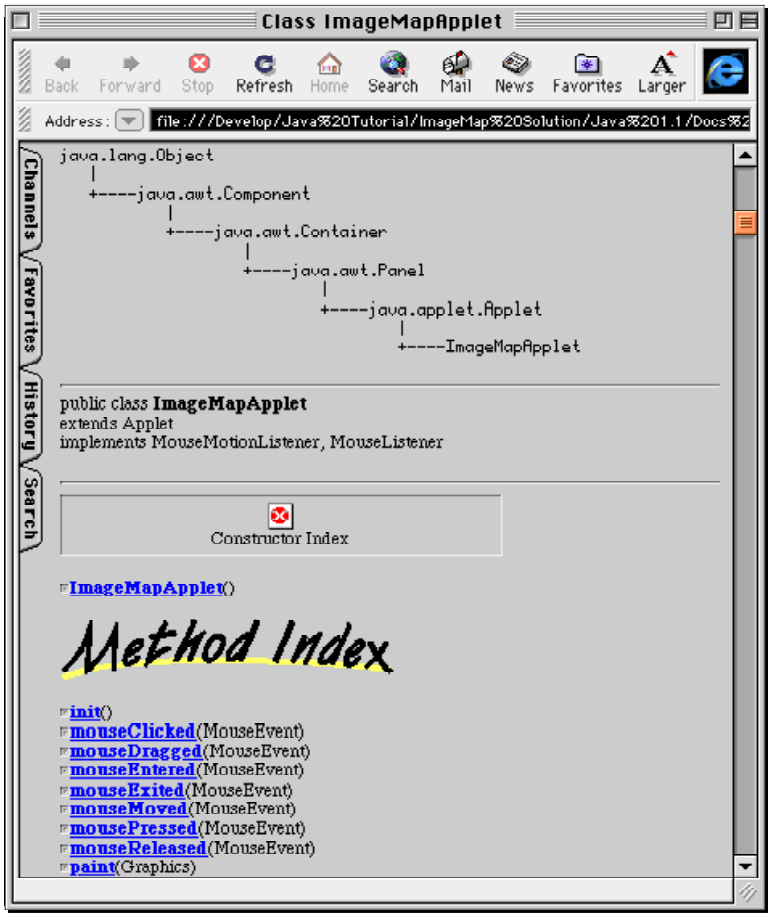
Figure 5.3 JavaDoc folder

The last step is to open the HTML files in your preferred HTML Browser (Figure 5.4). A separate HTML file is generated for every public Java class included in the target. There are also some default files generated depending on the structure of your Java project.

You may find some links do not work. This is because your project does not include this information. For example, if your project doesn't include packages, the link to `Packages.html` fails because the file was not generated by JavaDoc.

TIP: Images are not automatically generated by JavaDoc, though the HTML source that references these images is generated. You may create your own images to use, use the images from the Java API documentation, or you can use the images supplied with your CodeWarrior installation. The images are located in: `Metrowerks CodeWarrior:Java Support:Metrowerks:JavaDoc:images.`

Figure 5.4 ImageMap doc class



6

Target Settings for Java

CodeWarrior *target settings* are options you can specify to determine various aspects of your project's behavior, such as how it compiles and links. Choosing the right settings can significantly improve the size and speed of your final code.

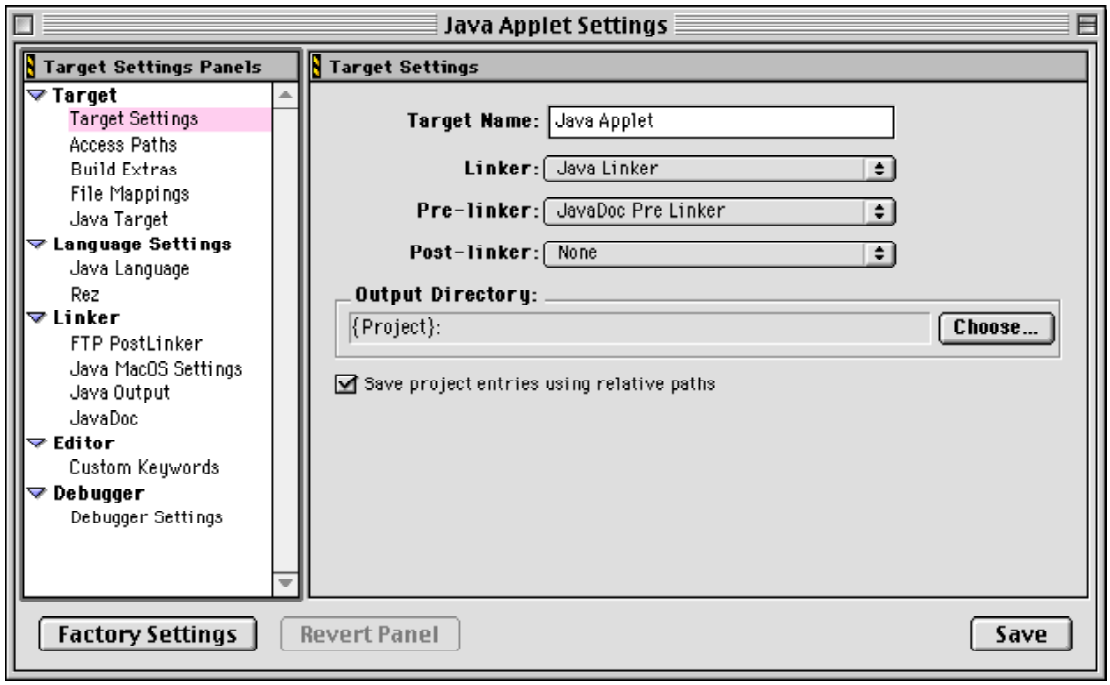
Target settings are organized into *panels* that you can display in CodeWarrior's project settings window. Different settings panels control various properties of the project. The Target Settings dialog box is shown in Figure 6.1.

This chapter discusses only those project settings panels that relate specifically to Java programming:

- Target Settings
- Java Target
- Java Command Line
- Java Language
- FTP Post Linker
- Java MacOS Post Linker
- Java Output
- JavaDoc

See the *IDE User Guide, C Compilers Reference,* and *Pascal Compilers Reference* for information about other settings panels available in the Target Settings Dialog.

Figure 6.1 Target Settings Dialog



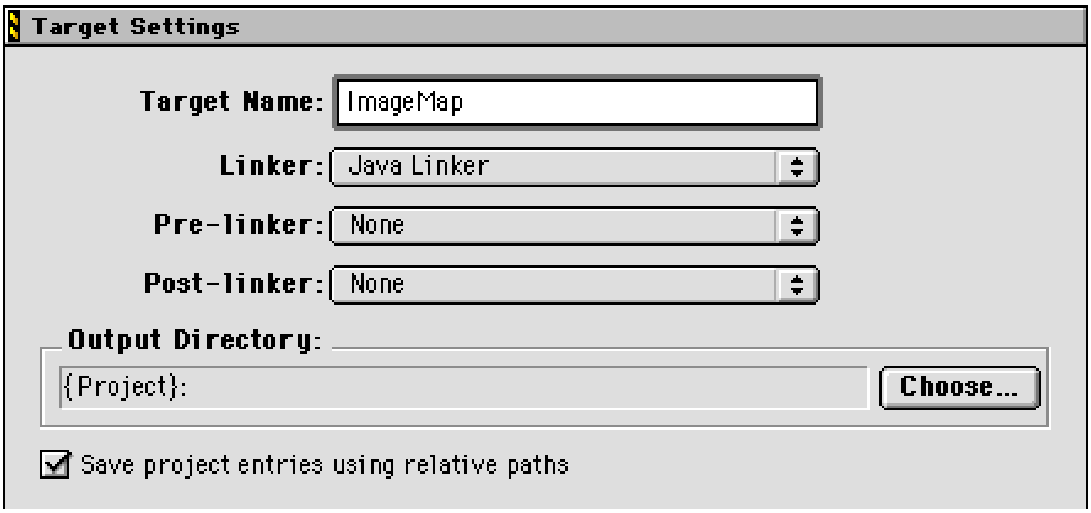
Target Settings

The **Target Settings** panel is the most critical panel in CodeWarrior. This is the panel where you pick your target operating system and/or microprocessor.

The **Target Settings** panel, shown in Figure 6.2, allows you to set the name of your target, as well as which linker and post linker plugins to use for the target. When you select a linker, you are specifying the target operating system and/or chip. The other panels available in the Settings dialog box will change to reflect your choice.

Because the linker choice affects the visibility of other related panels, you must set your target first before you can specify other target-specific options like compiler and linker settings.

Figure 6.2 Target Settings Panel



The items in this panel are:

Target Name	Post-linker
Linker	Output Directory
Pre-linker	Save project entries using relative paths

Target Name

Use the **Target Name** field to set or change the name of a target. When you use the Targets view in the Project window, you will be able to see the name that you have set.

This is not the name of your final output file. It is the name you assign to the target for your project file to use. The name of the final output file is set in the Java Output panel.

Linker

Choose a linker from the items listed in the Linker pop-up menu. There is one Java linker.

Pre-linker

Some targets have pre-linkers that perform additional work (such as a data format conversion) before linking. There is one pre-linker for Java: the JavaDoc Pre-Linker.

See also "JavaDoc."

Post-linker

Some targets have post linkers that perform additional work (such as a data format conversion) on the final executable. Based on this setting, corresponding panels will be added to the **Linker** section of the panel list on the left-hand side of the Target Settings dialog. For instance, if the **JCommand Line** linker is selected, a **Java Command Line** post linker panel will appear under the **Linker** section.

The post linkers available for your use are described in detail on the following pages:

FTP Post Linker "FTP Post Linker"

Java MacOS Post Linker "Java MacOS Post Linker"

JCommand Line "Java Command Line"

Output Directory

This is the directory where your final linked output file will be placed. The default location is the directory that contains your project file. Click the **Choose** button to specify another directory.

Save project entries using relative paths

Set this option if you have Java files that use the same name, but are physically located in a different directory. This setting allows the IDE to recognize it as a different file. If you do not have files with the same name, you can leave this option off.

Java Target

The settings you can specify on the **Java Target** panel depend on whether you're creating an Applet, a library, or Jar file.

This section describes settings available for each project type:

- Applet
- Application
- Library

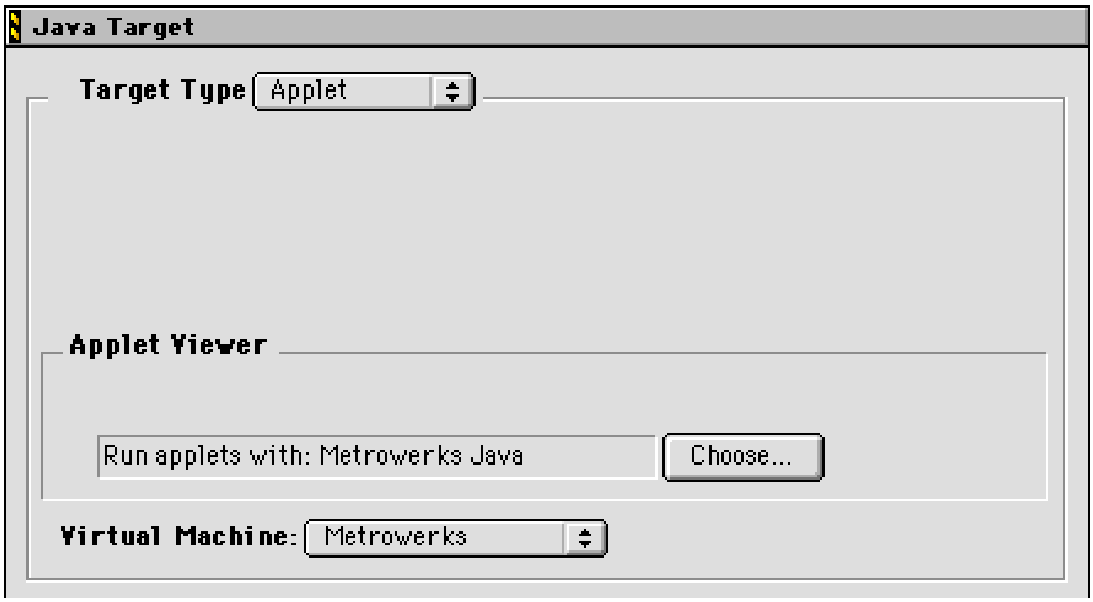
Applet

If the project **Type** is **Applet** (Figure 6.3), you have the choice of using a default applet viewer or choosing your own. Click the **Set** button to the right of the **Applet Viewer** edit field to choose a different applet viewer.

On the Mac OS, the default applet viewer is the Metrowerks Java application. If the Metrowerks Java application is the chosen applet viewer, you also have the option of setting the Virtual Machine to use. You can choose either: Metrowerks VM, Metrowerks - JIT, or Apple MRJ. If any of these are not installed, it will be grayed out in the menu.

For more on the Metrowerks Java application, see "Metrowerks Java."

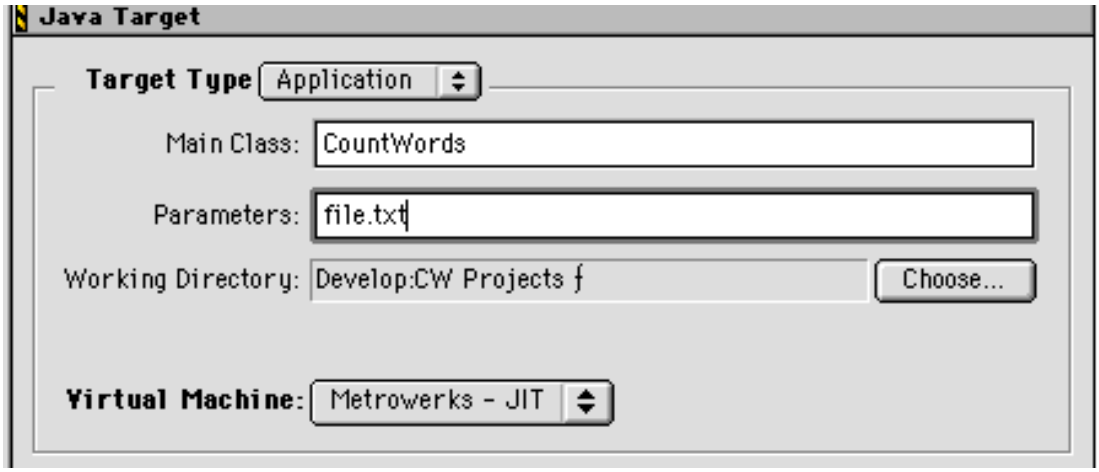
Figure 6.3 Applet project type



Application

If the project **Type** is **Application** (Figure 6.4), you need to specify your main class and any parameters your class requires. A working directory is optional and depends on the requirements of your class.

Figure 6.4 Application project type



Main Class

The **Main Class** field identifies the name of the class containing the `main()` method in a Java application. Enter only the name of the class, do not use the `.class` extension. You are specifying the class itself, not the object file.

CodeWarrior passes this name to the VM running the application so it can begin execution. This field must have a value in order to run a Java application.

NOTE: the `main()` method must be declared `public static void main(String args[])`, and must be in a public class. Also, the capitalization of the name in the **Main Class** field must match the capitalization in the Java file. In Java, `myclass` and `MyClass` are different classes.

Parameters

Enter any arguments you wish to pass to the main() method when the application launches. A series of arguments must be delimited by spaces.

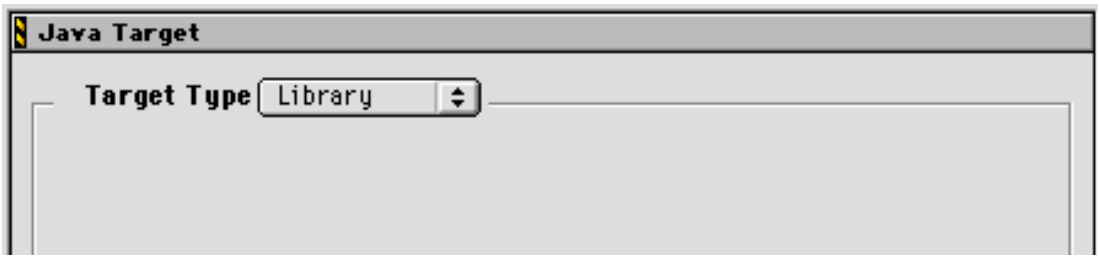
Working Directory

The **Working Directory** field is optional. A Java application sets the working directory to the directory where the VM resides. Set this field if you require a different working directory.

Library

A project Type of **Library** (Figure 6.5), does not have any further options to set.

Figure 6.5 Library project type



Java Command Line

The Java Command Line panel allows you to launch a Java application after your project is successfully built to do further processing of your project output.

For instance, you may use it to launch RMIC, a post linker that takes .class files and generates stub classes. The java linker would generate the .class files. Then the RMI compiler (RMIC) would take those class files and generate stub class files for use with RMI.

The panel contains the following items:

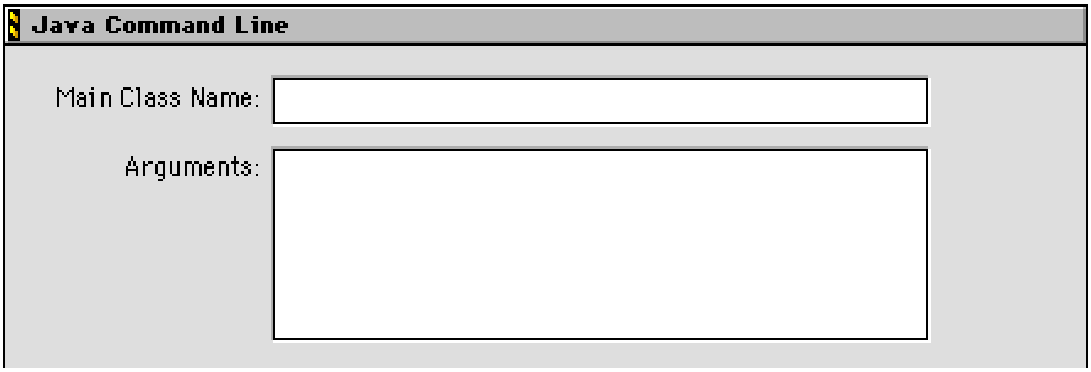
Main Class Name

Put the name of the main class of the application you want to be invoked into this field. Once your project is successfully built, the linker will run the application.

Arguments

Put any arguments to be sent to your application into this field.

Figure 6.6 Java Command Line panel



Currently, this linker adds paths to all zips and jars in the project to the VM classpath. It also adds the project output to the classpath.

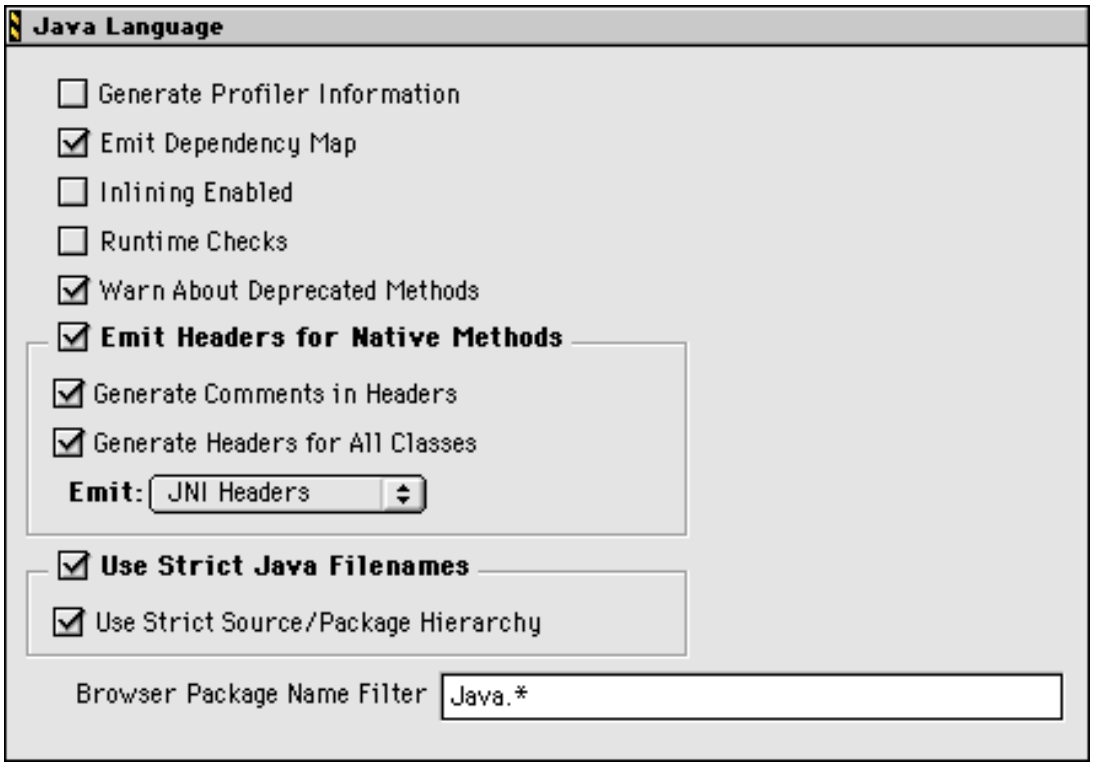
NOTE: Although this linker can execute any Java application specified in the command line settings, its intended use is to invoke java command line post linker type tools (such as RMIC, obfuscators, etc.). Therefore, there is no support for AWT based apps nor apps which make use of System.in.

WARNING! Since it is impossible for the post linker to know what type of application it will be invoking, it is the responsibility of the user to make sure that only the appropriate type of command line tools are run using it.

Java Language

The **Java Language** panel (Figure 6.7) contains project settings related to code generation for the Java platform.

Figure 6.7 Java Language panel



These options in this panel are:

Generate Profiler Information	Emit Headers for Native Methods
Emit Dependency Map	Use Strict Java Filenames
Inlining Enabled	Use Strict Source/Package Hierarchy
Runtime Checks	Browser Package Name Filter
Warn About Deprecated Methods	

Generate Profiler Information

To use this option, you must include `Profiler.zip` in your project. This option adds timing information to your Java methods. When your Java code finishes executing, a new file is created with the timing data. You can view this file with the MW Profiler application.

NOTE: You cannot profile Applets. Only Applications.

Emit Dependency Map

This option is similar to generating a link map in other languages such as C and C++. When enabled, a new file is created in the same directory as your project file called `sourcefile.JMAP`, where `sourcefile` is the name of your `.java` source file. The JMAP file lists all the class dependencies for each class in the file.

For example, the statement `import java.*` is common in Java source code. The dependency map tells you the *exact* class dependencies. So the statement `import java.*` may become:

```
import java.applet.Applet;
import java.applet.AppletContext;
```

You can then use these statements in your Java source files instead of importing everything. This makes things cleaner, faster and more portable to other Java compilers.

Inlining Enabled

Allows inlining of smaller methods where appropriate.

Runtime Checks

Please read the release notes for information on this option.

Warn About Deprecated Methods

Gives warning messages if you are using any depreciated methods.

Emit Headers for Native Methods

Emits headers for any native methods for *every* class in your project. When this option is enabled, the **Emit** popup menu appears allowing you to choose between **JNI Headers** and **Sun VM Headers** . JNI is the newest and preferred method. However, for compatibility reasons, you may wish to choose the **Sun VM Headers** option.

The other options that can be used in conjunction with Emit Headers for Native Methods are **Generate Comments in Headers** and **Generate Headers for All Classes** .

Generate Comments in Headers moves comments from the java sources files to the native headers.

Generate Headers for All Classes generates the headers in a tree (like sun/tools/java/Identifier.h) instead of mangled with the package (like sun_tools_java_Identifier.h)

Use Strict Java Filenames

Forces strict class names based on the Java source file names. For example, if you have a java source file called FooBar.Java, the compiler expects to find a class called FooBar.class in this file. No other class can be in this file.

NOTE: This option should be used in conjunction with Use Strict Source/Package Hierarchy.

Use Strict Source/Package Hierarchy

Forces strict adherence to source and package hierarchy paths. For example, if you have the following package:

```
Foo\bar\seam
```

The compiler expects a to find file `seam.java` in a directory called "`bar`" which is in a directory called "`Foo`."

This option is only available when Use Strict Java Filenames is enabled.

Browser Package Name Filter

Accepts a semicolon delimited list of names that you do not want to show up in the class browser. If you type in: `java.io; java.lang`, the browser will only display `File`, `Object` and `reflect.Method` for classes like `java.io.File`, `java.lang.Object`, and `java.lang.reflect.Method`.

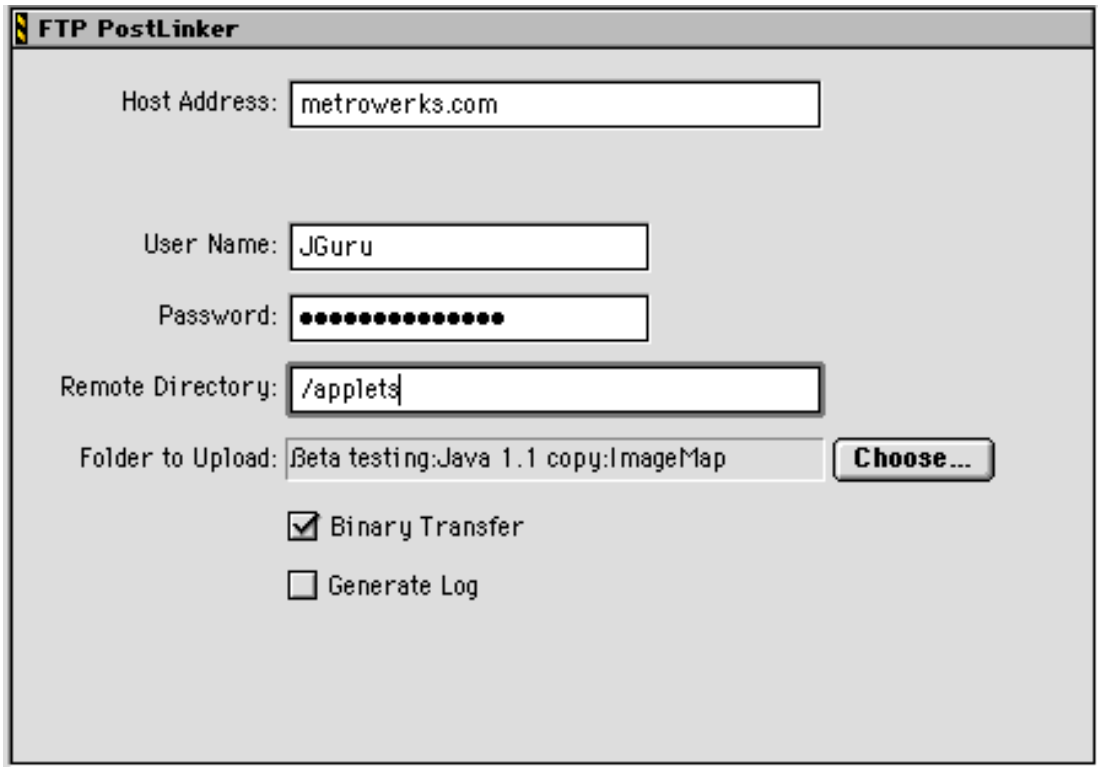
You can also use wildcards like `java.*`, in which case the browser will strip off the entire package for any class starting with "java."

Finally, you can also type in "`*`" (no quotes), in which case the browser strips all packages off everything.

FTP Post Linker

The **FTP Post Linker** panel allows you to move a folder containing your completed Java binary and any associated files to a server to be accessed through the World Wide Web.

Figure 6.8 FTP Post Linker panel



The items in this panel are:

Host Address	Folder to Upload
User Name	Binary Transfer
Password	Generate Log
Remote Directory	

Host Address

The host address of the server you want to upload your Java files to.

User Name

Your user ID on the host server.

Password

Your password on the host server.

Remote Directory

Directory you want to upload the Java files to on the host system. You must have access privileges for this directory.

Folder to Upload

The full path to the folder on your local hard drive containing the files to upload. There is no way to specify that individual files be uploaded, so the linker will transfer the entire contents of the folder specified here. Make sure you do not have any files that you do not want transferred in this folder.

Binary Transfer

Transfer the files to the host system using Binary mode instead of ASCII. It is recommended this option always be turned on to avoid any transfer problems.

Generate Log

Generates a text log file of the transfer. Any errors or problems are recorded in this file.

endrange marker for the FTP PostLinker panel.

Java MacOS Post Linker

The settings in the **Java MacOS Settings** panel control the operation of the Java MacOS Post Linker. This post linker determines how Mac OS Java applications are packaged.

The **Mac OS Java Output Type** popup menu can be set to one of the following settings:

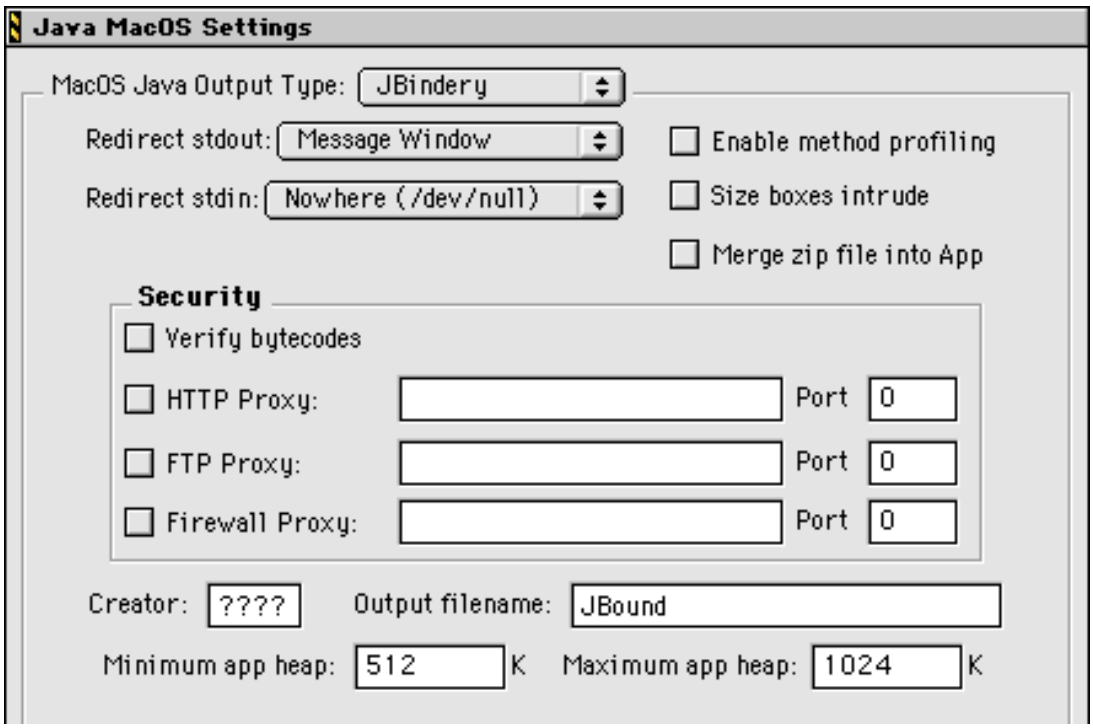
- JBindery
- MW Standalone
- Mac OS Zip

This section describes the settings for each option in detail.

JBindery

When the **JBindery** output type is selected, the panel appears as in Figure 6.9. JBindery is an application that you use to package or execute Java™ applications on the Mac OS platform.The settings contained in this panel allow you to control how JBindery builds your application.

Figure 6.9 Output as JBindery



The Java Mac OS Post Linker will launch JBindery using the follwing settings.

Redirect stdout

This popup menu allows you to redirect stdout. You can either redirect stdout to a message window generated by JBindery, or you can instruct JBindery to ignore stdout altogether.

Redirect stdin

This popup menu allows you to redirect stdin. You can either redirect stdin to a message window generated by JBindery, or you can instruct JBindery to ignore stdin altogether.

Enable method profiling

Check this checkbox to enable method profiling.

Size boxes intrude

This checkbox determines whether the size box should appear within the corner of the actual window. If the size boxes intrude, any Abstract Window Toolkit components that would normally appear under the size box are not drawn. If you do not check this box, an extra strip of empty space is added to the bottom of the window to accomodate the size box.

Merge zip file into app

If this checkbox is checked, the zip file will be merged into the final application when you build your project.

Verify Bytecodes

Check this checkbox if you want the code verifier to check local Java bytecodes before execution. If this checkbox is not selected, JBindery will still automatically check any bytecodes obtained from a remote source (such as over a network).

HTTP Proxy

If you would like your application to use an HTTP proxy server, check this checkbox, and supply the name and port ID of the proxy server in the corresponding edit fields.

FTP Proxy

Check this checkbox if you would like your application to use an FTP proxy server. Supply the name and port ID of the proxy server in the corresponding edit fields.

Firewall Proxy

This option specifies whether you want to use a firewall proxy. When this checkbox is checked, the Java application uses the firewall proxy server specified in the name and port edit fields when accessing servers outside the security firewall.

Creator

This edit field lets you specify the creator of your final Java application.

Output Filename

This edit field lets you specify a creator for your Java application. This unique string identifies the application and any documents that the application may create. If you plan to publically distribute your application, you must register its creator name with Apple through Developer Technical Support to avoid collisions between names used by different developers.

You can register a creator online or view currently registered creators at the following Web site:

http://devworld.apple.com/dev/cftype/main.html

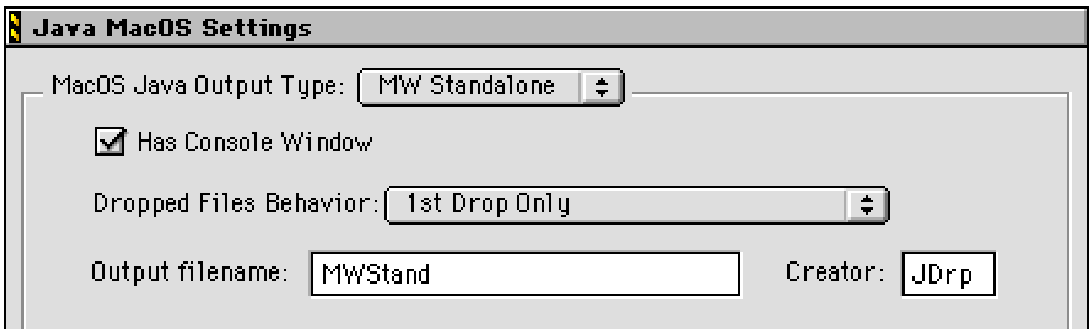
Minnimum and Maximum app heap

These two edit fields let you specify the amount of memory to use when executing this Java program. Mac OS Runtime for Java uses temporary memory for most allocations (the Java virtual machine and so on) so an application heap of 512K is usually sufficient.

MW Standalone

When the **MW Standalone** output type is selected, the panel appears as in Figure 6.10. This creates a **MW Standalone** double-clickable MacOS java application.

Figure 6.10 Output as MW Standalone



Has Console Window

If this checkbox is checked, MW Standalone will add a console window to your application.

Dropped Files Behavior

This popup menu has four items which determine the behavior of your application when files are dropped onto it:

No Finder Drops — If this menu item is chosen, files dropped onto your application will not be added to the application's argument list. Your application may be started up by a dropped file, but the file will be ignored. All subsequent dropped files will also be ignored. As a result of this behavior, only one instance of the JVM will be created during the application's lifetime.

1st Drop Only — If this menu item is chosen, only the first file dropped onto your application will be added to the application's argument list. In order for a file to be recognized, the application must not be running when the file is dropped onto it. Any files dropped onto your application once it is running will be ignored. This allows for only one instance of the JVM to be created during the application's lifetime.

Multiple Drops, Kill JVM for each — If this menu item is chosen, each file dropped onto your application will cause the application to be terminated if it is running, and a new instance of your applica-

tion to be created. This allows only one instance of the JVM to be exist during the application's lifetime.

Multiple Drops, New JVM for each — If this menu item is chosen, each file dropped onto your application will cause the application to be terminated if it is running, and a new instance of your application to be created. This allows multiple instances of the JVM to exist at one time during the application's lifetime. This was the previous behavior of MW Standalone applications.

Output filename

This edit field determines the name of the application being created.

Creator

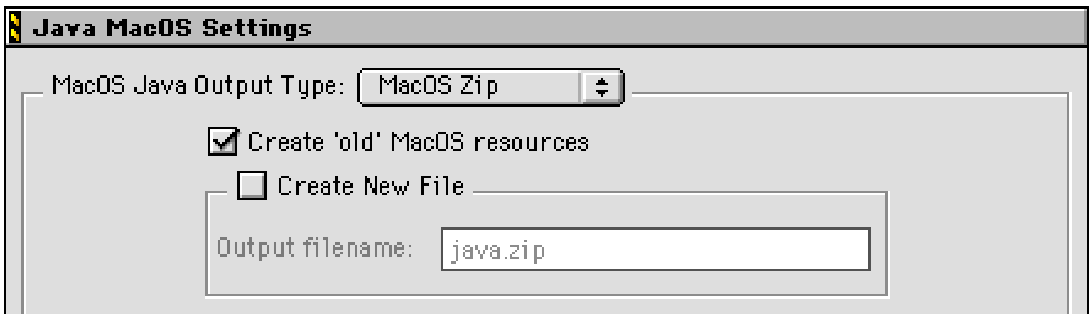
Supply the four-character creator code for the application in this edit field.

For more information on how to use **MW Standalone** , refer to "StandAlone Applets".

Mac OS Zip

When the **Mac OS Zip** menu item is selected, the panel appears as in Figure 6.11. Previous versions of the Java Linker copied Mac OS resources into the output zip file. This was handy for adding 'vers' resources to your zip file, for example. This option allows you to do the same thing.

Figure 6.11 Output as Mac OS Zip



Create 'old' MacOS resources — The Java Linker used to create certain resources in the output zip file that were used to assist in building standalone applications. These resources are no longer needed by CodeWarrior, and are no longer written. However, with JBindery in MRJ SDK 2.0.1, if you drop jar/zip files containing these resources onto JBindery, it will still parse out the main class name and the main arguments and use them. This option is available for this reason.

Create New File — If this checkbox is checked, resources will be added to a copy of the Java Linker output file, rather than the original output file. Otherwise, resources will be merged into the Java Linker output file. Supply the name of the new file in the **Output filename** edit field.

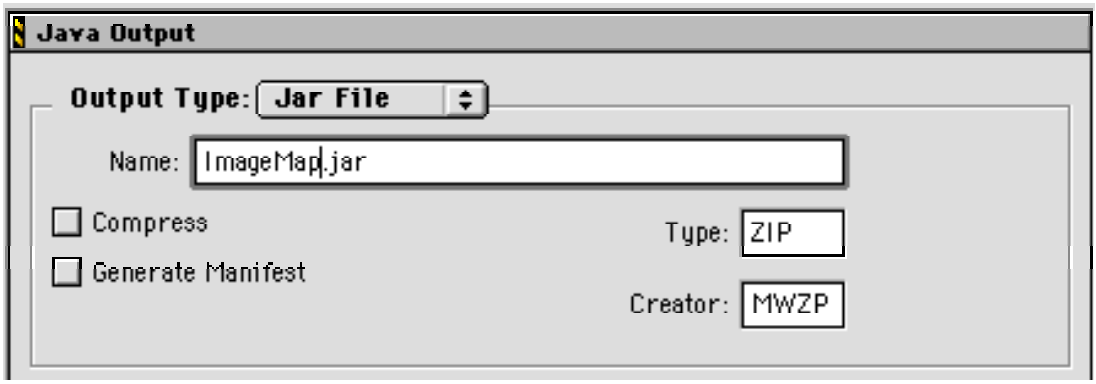
Java Output

The settings on the **Java Output** panel (Figure 6.12) control the operation of the CodeWarrior linker for Java. The **Output Type** can be:

- Class Folder
- Jar File

This section describes the settings for each option in detail.

Figure 6.12 Java Output panel

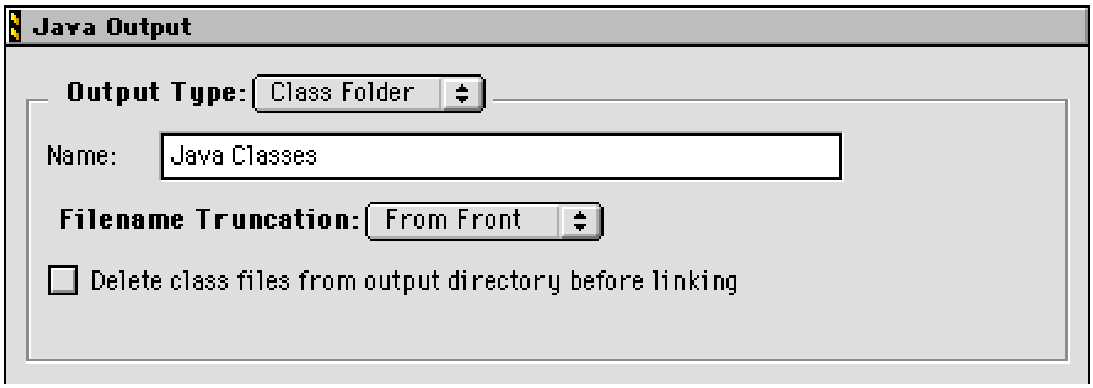


Class Folder

If you choose **Class Folder** from the **Output Type** popup menu (), the IDE will create a new folder with the name you specify in the **Name** field. Every class in your project will be placed in this folder. The Mac OS has a 31 character maximum for filenames. The **Filename Truncation** menu controls where the IDE will remove characters of the final class file name if it's too large. You can choose **From Front** , **From Middle** , or **From End** .

Delete class files from output directory before linking simply removes all class files in the target directory before linking your code. This is useful if you recompile your source often as it makes sure no old code exists in the target folder.

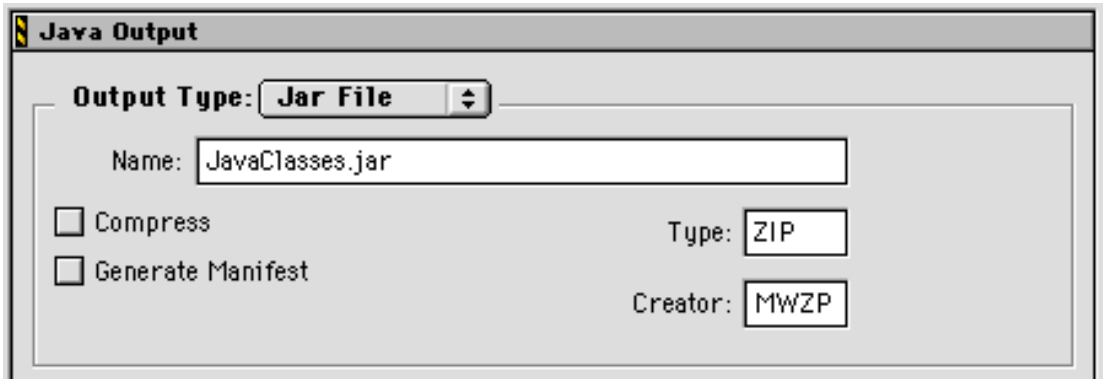
Figure 6.13 Output as class folder


Java Output
Output Type: Class Folder
Name: Java Classes
Filename Truncation: From Front
Delete class files from output directory before linking

Jar File

A Jar file is a Java Archive file. The **Jar File** option (Figure 6.14) allows you to control how the Jar file is created. Choose **Compress** to create a compressed Jar file. Choose **Generate Manifest** to create manifest information in the Jar file.

Figure 6.14 Output as Jar file



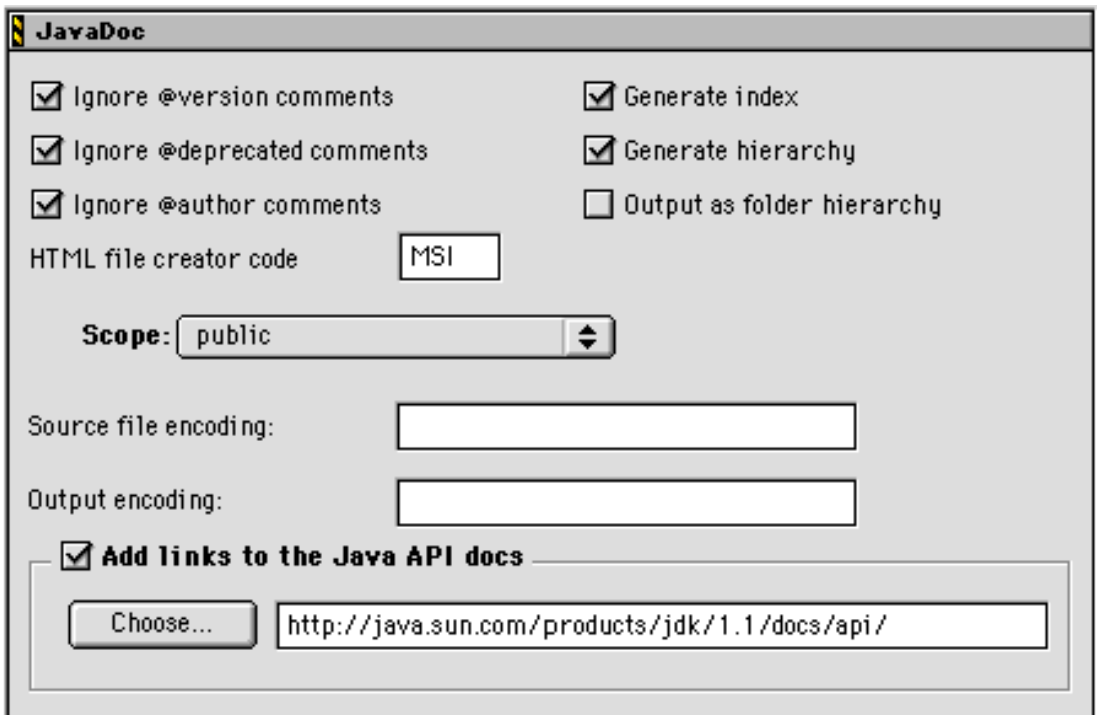
The **File Type** and **Creator** fields allow you to change the default application used to open them if you double-click on the file. The default settings open the Jar file in Class Wrangler.

See also "Using Class Wrangler."

JavaDoc

The **JavaDoc** panel (Figure 6.15) controls how JavaDoc creates documentation for your Java source.

Figure 6.15 JavaDoc panel



Ignore options

Three ignore options: **Ignore @version comments** , **Ignore @deprecated comments** , and **Ignore @author comments** tell JavaDoc to omit comments with these "tags" from the final output when enabled.

Generate Mac-friendly filenames

Tells JavaDoc to generate filenames shortened to 31 characters to create cross-platform friendly HTML documents.

Generate index

Generate index tells JavaDoc to create index information in the final output. Sometimes, the index can be very large. Turn this option off if you do not want index information generated for your project.

Generate hierarchy

Generate hierarchy tells JavaDoc to generate class hierarchy information. Turn this option off if you do not want hierarchy information created.

Output as folder hierarchy

Instead of only outputting the HTML files for a project to a flat folder hierarchy with extremely long filenames, the JavaDoc Pre-Linker has the option of putting the files into a package-based folder hierarchy. The actual file names are only the class name.

HTML file creator code

Use this option to set the creator code for HTML files to your favorite HTML browser.

Scope

The **Scope** popup menu controls which classes will be included in the documentation. Choices include:

- public
- protected and public
- protected, public and package
- all

NOTE: If you do not specifically declare a class to be public, private, or protected, it will automatically be declared as package.

Encoding options

The two encoding options: **Source file encoding** and **Output encoding** are for internationalization of your JavaDoc files. For example, the Java source files can be in Arabic and output in English.

Encoding numbers are used in each field. JavaDoc converts the number in the source field to Unicode, and then from Unicode to the value specified in the output field. Leave these fields blank to use the default ISO Latin (859-1) encoding.

For more information on encoding, see:

`http://java.sun.com/products/jdk/1.1/intl/html/intlspec.doc7.html`

Add links to the Java API docs

This option adds links to the main Java API docs where appropriate in your code. The default link is on the Web. If you have a dial-up connection, you might prefer to have the API docs somewhere on your local machine. This way, you can specify the URL to be something like `file:///my_drive/jdk_docs/`. You can also specify a relative URL.

NOTE: If you move the docs around, the links may break.

7

Debugging Java Projects

The CodeWarrior debugger lets you debug Java code as easily as C/C++ or Pascal code. This chapter assumes you're already familiar with the CodeWarrior debugger. If you need more information about the debugger, see the *Debugger User Guide*. If you want a quick tutorial showing you how to use the debugger with Java, see "Programming Tutorial for Java." If you want step-by-step information on running a Java project under the debugger, see "Debugging a Java Project."

This chapter contains the following sections:

- Debugger Features and Limitations describes some of the features in the debugger that help you debug Java code and some limitations of the debugger.
- Special Debugger Features for Java goes into more detail about some special debugger features for Java.

Debugger Features and Limitations

The CodeWarrior Debugger contains many features that are specifically for debugging Java projects. It lets you:

- Debug Jar files, class files and applications.
- View a disassembly of the Java Virtual Machine instructions for your class, described in "Viewing the Java Virtual Machine Disassembly."
- Debug multiple thread programs, as described in "Debugging Threads."
- View the Java Virtual Machine registers, described in "Viewing Java Virtual Machine Registers."

- Break on Java exceptions, described in "Breaking on Java Exceptions."

The debugger has a couple limitations that a future version will solve:

- Multi-language debugging is not fully supported. If you implement some native methods in C, you can open both the Java class files and the C SYM files for the native methods in the debugger at the same time. And if you set breakpoints in both the C and Java files, the debugger will automatically break at those statements. However, the stack crawl window does not show the call chain across languages and you cannot single step from C code to Java code or from Java code to C code.
- The debugger cannot debug compressed Jar or compressed zip files. This includes Java applications that are in compressed Jar or compressed zip format.

Special Debugger Features for Java

This section explains some of the special CodeWarrior Debugger features for Java. It contains the following:

- Breaking on Java Exceptions
- Opening Multiple Class Files in One Browser
- Choosing a Java Applet Viewer for Debugging
- Debugging Threads
- Viewing Java Virtual Machine Registers
- Viewing the Java Virtual Machine Disassembly
- Specifying Java Debugger Settings

Breaking on Java Exceptions

The CodeWarrior debugger can automatically break when your class throws an exception. Just choose one of these options from the **Break on Java exceptions** hierarchical menu in the Control menu.

If you choose	The debugger breaks on exceptions from…
All exceptions	Any class, including the Java API classes.
Exceptions in targeted classes	Only those classes in the file that you opened.
No exceptions	No classes.

Opening Multiple Class Files in One Browser

If you're debugging a project that creates multiple class files, you can view all the class files for a folder in one Browser window. Just turn on the **Open all class files in directory hierarchy** option in the Symbolics preference panel in the CodeWarrior Debugger. When you open a class file, the Browser window displays the classes for that file as well as all for the files in the same folder and its subfolders.

If that option is off, the Browser window displays the classes for one class file only.

Choosing a Java Applet Viewer for Debugging

When debugging a Java applet, the applet runs within an applet viewer or a Java-aware HTML browser like such as Internet Explorer. To learn how to specify the applet viewer, see "Applet."

When debugging a Java applet, the viewer you choose must support one of the following Virtual Machines:

- Metrowerks Java—Metrowerks Java runtime environment and applet viewer
- Apple MRJ—uses the Apple Applet Runner, which is part of Apple's Mac OS Runtime for Java (MRJ).

NOTE: Although many applications can run an applet, you can only use these to run an applet under the Metrowerks debugger.

That's because the debugger requires the application to support Metrowerks's Java debugging API, and these are the only Virutal Machines that do so at this time.

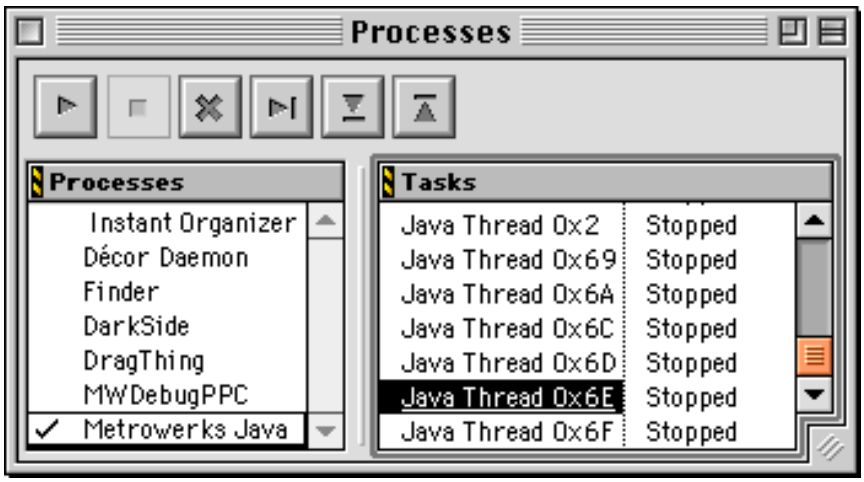
If the Java program is an application, the debugger always uses Metrowerks Java as the applet viewer.

Debugging Threads

When you are debugging a Java program, the running process is either the specified applet viewer, or the Java interpreter, such as Metrowerks Java.

You can view all the threads in a Java program in the Processes window. Just choose **Show Processes** from the Window menu and select the Metrowerks Java process in the left pane. The Java threads appear in the pane to the right.

Figure 7.1 The Processes window



To display a Program window for a thread, double-click it. To pause, continue, or kill a thread, select it and use the buttons in the Processes window.

NOTE: Many of the threads listed in the processes window are spawned by the applet viewer or interpreter running the Java program, or threads for the CodeWarrior Debugger. Pausing or killing one of them will have unforeseeable (and possibly quite unfortunate) consequences.

Viewing Java Virtual Machine Registers

You can view the registers in the Java Virtual Machine. When a class is running under the debugger, choose **Show Registers** from the Window menu. It displays a window like Figure 7.2:

Figure 7.2 The Registers window for Java

Table 7.1 describes what each register means.

Table 7.1 Register meanings

This…	is the register that contains a pointer to…
PC	The currently executing statement.
optop	The top of the operand stack.
frame	The execution environment of the current method.
vars	The first local variable of the current method.

Viewing the Java Virtual Machine Disassembly

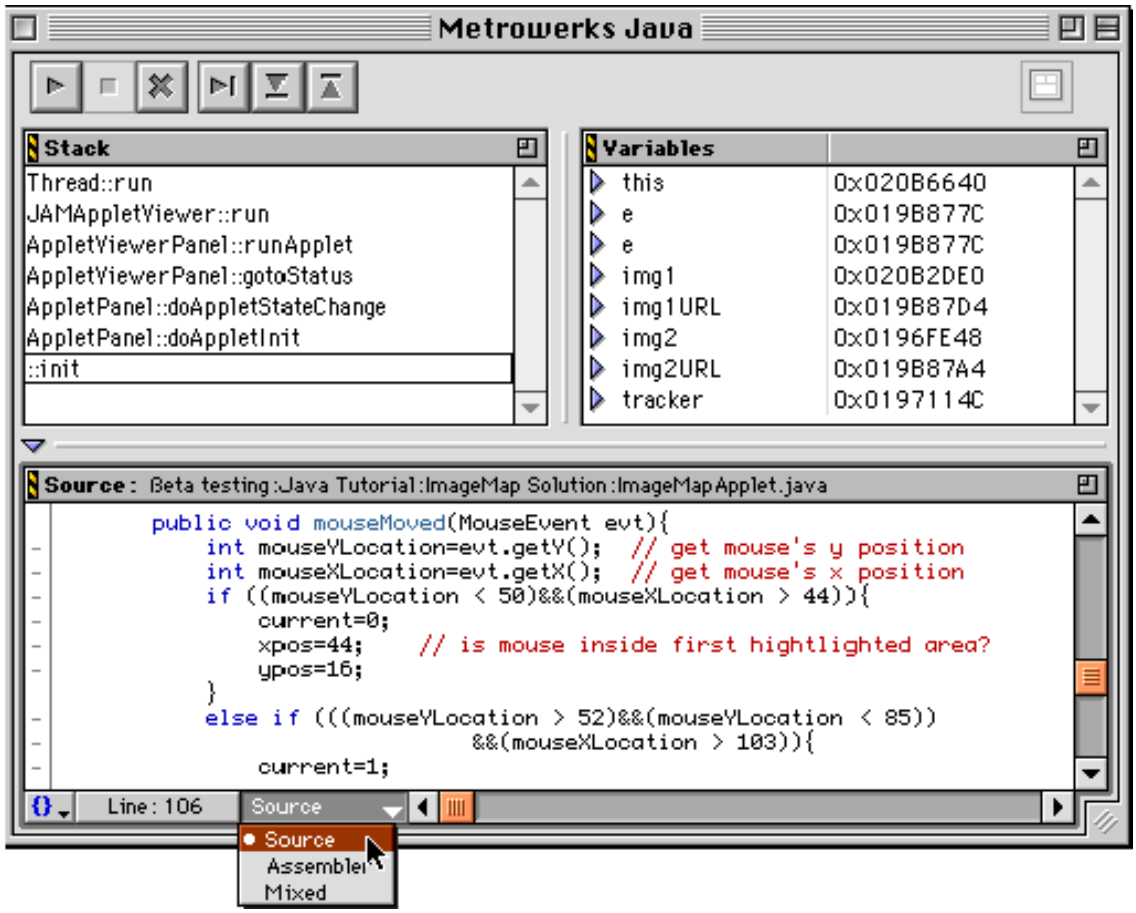
You can view a listing of the Java Virtual Machine instructions that implement the class. Just choose **Assembler** or **Mixed** from the

Source pop-up menu at the bottom of any Browser or Program window (Figure 7.3).

The mixed view shows your Java source first, then the assembly language instructions that make up the Java code immediately after.

See also "Disassembling Classes."

Figure 7.3 Viewing Java assembly



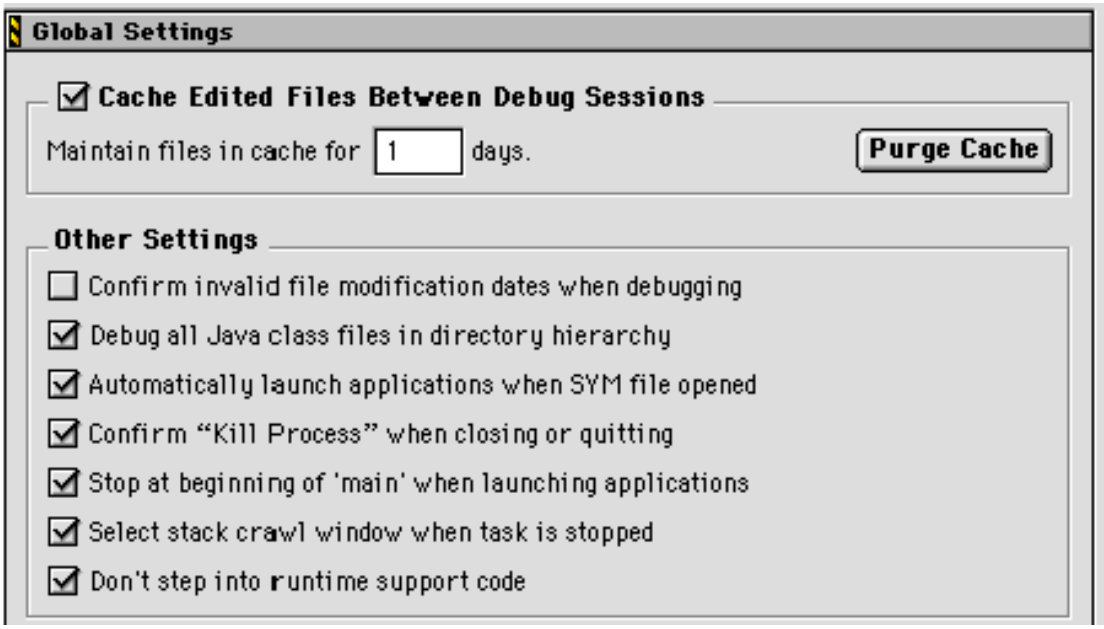
When viewing the assembly, you can still set breakpoints, step through code, and view variables. You can also view the assembly for a Java file in CodeWarrior by choosing **Disassemble** from the Project menu.

Specifying Java Debugger Settings

There is one setting in Global Preferences panel of the debugger group in the main IDE Preferences dialog that applies to Java: **Debug all Java class files in directory hierarchy** .

There are settings in the debugger that apply to Java: **Debug all Java class files in directory hierarchy** in the Global Preferences panel, and the settings in the Java Settings panel.

Figure 7.4 Symbolics preference panel


Global Settings
Cache Edited Files Between Debug Sessions
Maintain files in cache for 1 days.
Purge Cache
Other Settings
Confirm invalid file modification dates when debugging
Debug all Java class files in directory hierarchy
Automatically launch applications when SYM file opened
Confirm "Kill Process" when closing or quitting
Stop at beginning of 'main' when launching applications
Select stack crawl window when task is stopped
Don't step into runtime support code

The **Debug all class files in directory hierarchy** option opens all the class files in the directory and all contained directories, and merges them all together in the same Browser window.

8

Using Class Wrangler

Class Wrangler is Metrowerks' utility program for working with Java Archive (JAR) files. It allows you to add or remove files, as well as copy files from one zip/Jar file to another. Class Wrangler has other useful features as well.

NOTE: Class Wrangler is not a general purpose utility for compressed zip files. It will only handle zip archives that are all contained in one file (no disk-spanning files). If the zip archive is compressed, the deflate algorithm must be the one used in JDK 1.1 JAR files. Class Wrangler cannot handle encrypted files.

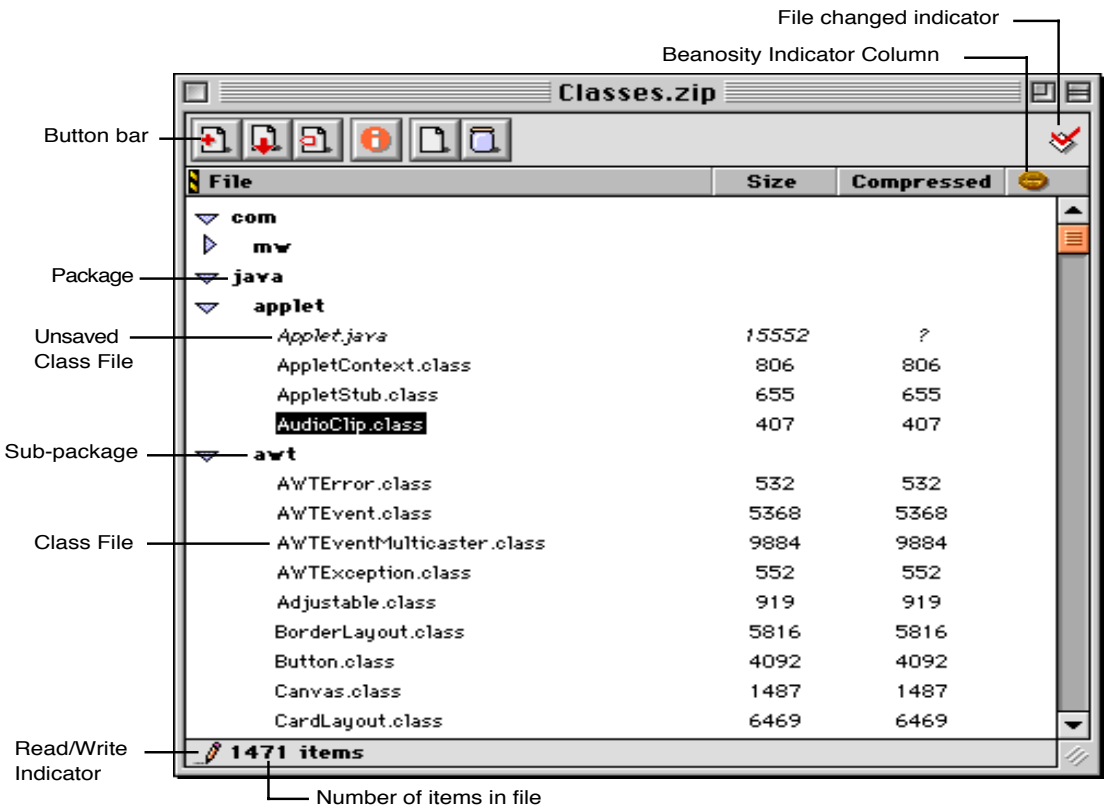
This chapter explains the Class Wrangler interface, and how you can use it to get your work done. The sections in this chapter are:

- Class Wrangler Window—what you see in the Class Wrangler interface
- Working with Files and Archives—discusses how to work with files and zip archives
- Editing Manifest Files—describes how to use the **Edit Manifest** command
- Class Wrangler Preferences—shows you how to control the way Class Wrangler works for you
- Comparing Archives—describes how to compare two Jar files

Class Wrangler Window

Class Wrangler uses a single window to display each package and class file stored in a zip archive or Jar file, as shown in Figure 8.1.

Figure 8.1 The Class Wrangler Window



You can use the button bar to add, extract, delete, and get information on files. Table 8.1 describes each button.

Table 8.1 Class Wrangler button bar

Button	Description
	Add File to Jar file.
	Extract Item and save item to disk.
	Delete item from Jar file.
	Get Info on class file. If a package or sub-package is selected, will get info on every file in the package and display the information separate windows, one for each class.
	Compression indicator. Clicking this button does not compress the file, rather it marks the file for compression (or uncompression) the next time the file is saved. A compressed file will occupy less disk space, but may require more time to load and process.
	Generate Manifest. This button indicates whether a Jar file has a manifest file or if one will be generated the next time the file is saved. You can toggle the bean status of an item in the Jar file by clicking in the JavaBean column. If the Jar file has a manifest, and a particular item is a bean, Class Wrangler draws a dot in the JavaBean column. Save the file to retain the changed state.

The diamond icon at the top right is empty if the zip archive hasn't changed since the last time it was saved, and contains a red check mark if the file has been changed.

Each package lists all the files and sub-packages it contains. To show or hide the contents of a package, click the disclosure triangle.

To the right of each file is its size in bytes. If a file is in italics, it has recently been added to the archive and hasn't been saved to disk yet. To save it, choose **Save** from the File menu.

You can use the **Copy** command in the Edit menu to copy the names of the selected files and packages to the Clipboard. When you copy a file, its fully qualified name is placed in the Clipboard. For example, if you copy `AudioClip.class`, this text is placed in the Clipboard:

```
java.applet.AudioClip
```

If you hold down the Option key while copying a class, an `import` statement for the selected class is placed in the clipboard:

```
import java.applet.AudioClip;
```

If you copy a package, its name, the names of its subpackages, and the names of its class files are placed in the Clipboard. If you hold down the Option key while copying, an `import` statement for the subpackages, and the class files are placed in the Clipboard.

Typing a full or partial file or package name selects the closest matching item in the Class Wrangler window. This is called "type ahead selection" and allows you to move around the archive quickly when you know the file or package you are looking for.

See also "Use full package name for type ahead."

Working with Files and Archives

This section explores how you can get work done using Class Wrangler. To avoid confusion, these topics refer to class *files*, zip *archives*, and Jar files (Java Archive files) consistently. Keep in mind that each is really just a kind of file.

Zip archives and Jar files are used interchangeably. The main difference is a Jar file can have manifest files, whereas Zip archives do not.

The topics in this section are:

- Opening a Zip Archive
- Creating a Zip Archive
- Adding Files
- Using the Add Files Dialog
- Add Directory
- Extracting Files
- Deleting Files
- Getting Information on Files
- Moving Files Between Archives

Opening a Zip Archive

There are two ways to open a zip archive:

- Drag and drop a zip archive onto the Class Wrangler icon.
- Choose **Open** from the File menu and choose the zip archive.

Class Wrangler opens the file and displays its contents in a window.

At the bottom of the Open dialog, there are three check boxes that let you choose which files the dialog displays. When no option is checked, the dialog displays only files whose Macintosh file types are for zip or class files. When the **Filter using file extension** option is on, the dialog also shows any file whose name ends in `.zip` or `.class`. When the **Filter using file contents** option is on, Class Wrangler examines the internal contents of each file and displays any file that it determines to be a zip or class file. This option is the most accurate, but takes the most amount of time. When the **Show all files** option is checked, Class Wrangler displays all files in a directory. This option overrides the other two, even if the other two are checked.

You can set these options generally for the program as well. See "File Filtering Settings."

Creating a Zip Archive

There are three ways to create a zip archive with Class Wrangler. To create an empty zip archive, choose **New** from the File menu. To create a zip archive that contains one class file, choose **Open** from the File menu, and select the class file. Class Wrangler displays a new window that contains that file.

To create a zip archive for several class files, follow these steps:

1. Make sure that Class Wrangler isn't running or that there are no open windows in Class Wrangler.
2. Drag and drop the class files onto the Class Wrangler icon.

Class Wrangler creates a window that contains those class files.

NOTE: If you drag class files onto the Class Wrangler icon while there is an open archive, the class files are added to the archive displayed in the frontmost Class Wrangler window.

Adding Files

Class Wrangler gives you three ways to add files to an archive.

- Choose **Add Files** from the Package menu or click the **Add Files** button in the Class Wrangler window. Class Wrangler displays a dialog box that lets you choose a group of files to add at once. For more information, see "Using the Add Files Dialog."
- Drag and drop class files onto the Class Wrangler icon. Class Wrangler adds them to the frontmost window.
- Drag and drop class files onto a Class Wrangler window. Class Wrangler adds them to the window.

If you try to add a class file that is already in the zip archive, Class Wrangler displays an alert asking you whether you want to overwrite the existing file.

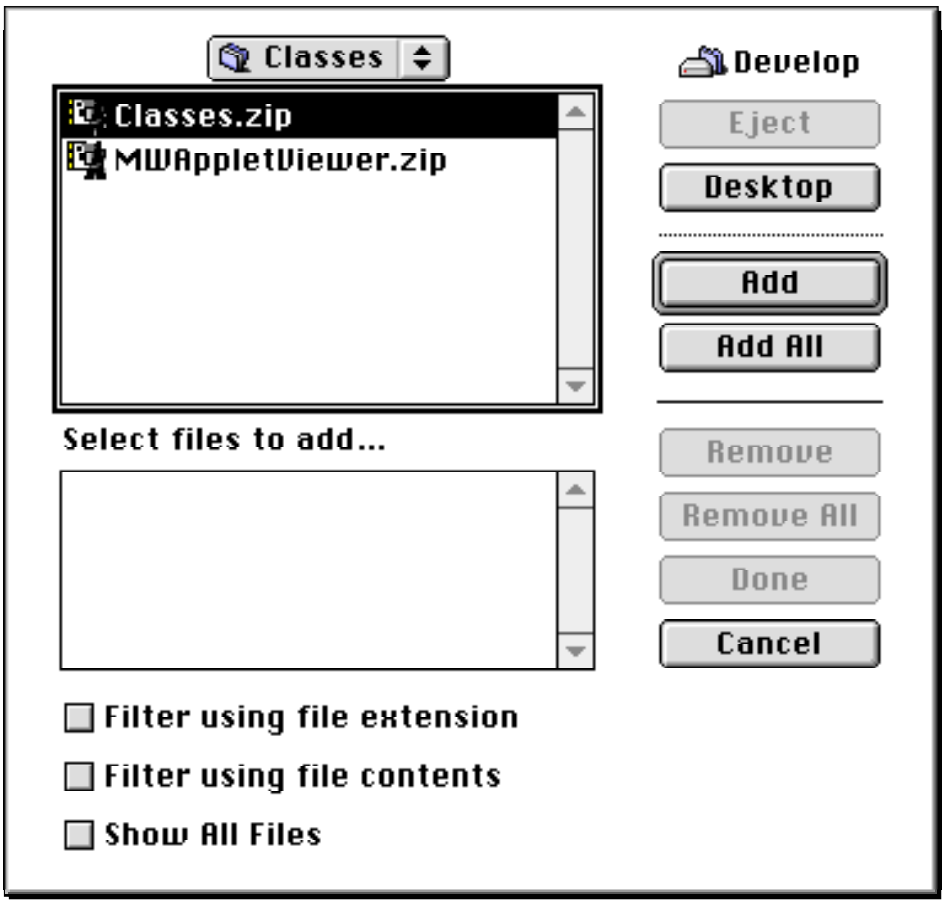
After you add class files, their names appear in italics in the window to show the new files have not been saved to the zip archive on disk. To save the new files to disk, choose **Save** from the File menu.

Using the Add Files Dialog

If you use the **Add Files** command or button, Class Wrangler displays a dialog like the one shown in Figure 8.2.

The top list is part of a standard file dialog and displays the contents of a folder. The bottom list contains all the files that will be added to the zip archive when you click the **Done** button. To choose a file to add, select it in the top list and click **Add**. To add all the files in the top list, click **Add All** . To remove a file from the bottom list, select it and click **Remove** . To remove all the files from the bottom list, click **Remove All** .

Figure 8.2 The Add Files dialog



The three check boxes at the bottom of the dialog let you choose which files the dialog displays. When no option is checked, the dialog displays only files whose Macintosh file types are for zip or class files. When the **Filter using file extension** option is on, the dialog also shows any file whose name ends in `.zip`, `.jar`, or `.class`. When the **Filter using file contents** option is on, Class Wrangler examines the internal contents of each file and displays any file that it determines to be a zip or class file. This option is the most accurate, but takes the most amount of time. When the **Show all files** option is checked, Class Wrangler displays all files in a directory.

This option overrides the other two, even if the other two are checked.

You can set these options generally for the program as well. See "File Filtering Settings."

Add Directory

The **Add Directory** command in the Package menu, allows you to add a package to the frontmost archive as shown in Figure 8.3. You must type the full path name. The directory is created in the archive.

Figure 8.3 Add Directory dialog

Extracting Files

Class Wrangler gives you two ways to extract class files from a zip archive. The class files remain in the zip archive after they're extracted. The process of extraction does *not* remove the class file from the zip archive.

- Select the files and choose **Extract Files** from the Package menu or click the **Extract Files** button in the Class Wrangler window. Class Wrangler displays a file dialog asking you where to place the files. Select a location and click OK. Class Wrangler places the class files in that location.
- Drag the files from the Class Wrangler window onto the Finder desktop. Class Wrangler places the files in that location.

TIP: If you don't want the folder hierarchy, hold down the Option key while dragging.

When Class Wrangler extracts a file, it places the file in a folder hierarchy that matches the file's package hierarchy. For example, if the file is in the hierarchy `java.util`, Class Wrangler creates the folder `java` at the location you selected, creates the folder `util` inside `java`, and places the file in `util`.

Deleting Files

To delete a class file from a zip archive, select the file and do one of the following:

- Choose **Delete File** from the Package menu
- Click the **Delete File** button in the Class Wrangler window
- Press Option-Delete

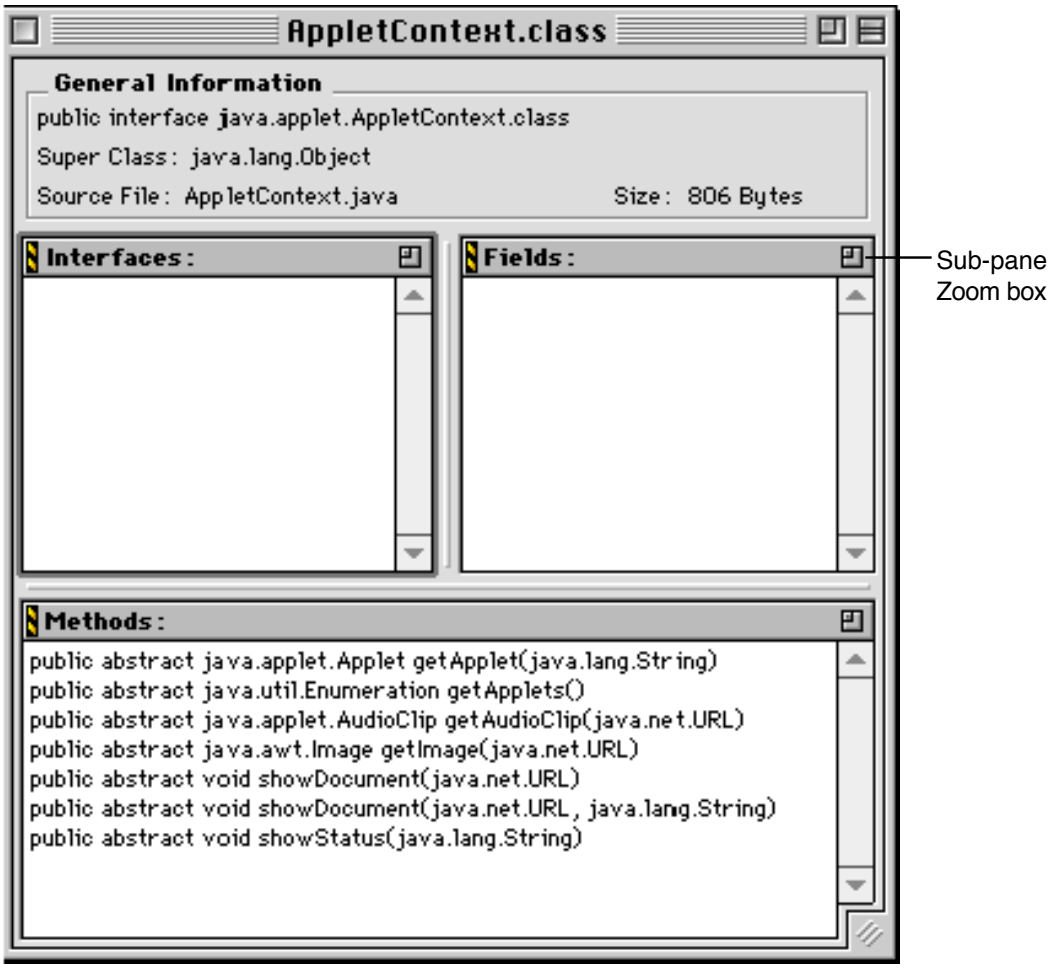
Class Wrangler indicates the file is flagged for deletion by displaying the file name in italics. You must choose **Save** from the File menu to completely remove the files.

Getting Information on Files

You can see information on any class file in a zip archive. Just select the class file and choose **Get Info** from the Package menu or click the Information button in the Class Wrangler window. Class Wrangler displays a window of information on the public class in the class file, including information on its fields, methods, interfaces, and superclass, as shown Figure 8.4.

Information windows are associated with an open archive document. If the associated archive document is closed, all of its information windows are closed.

Figure 8.4 The Class Wrangler Get Information window



Click the sub-pane zoom box control to expand the pane to fill the entire information window, or shrink the pane to see all sub-panes.

Moving Files Between Archives

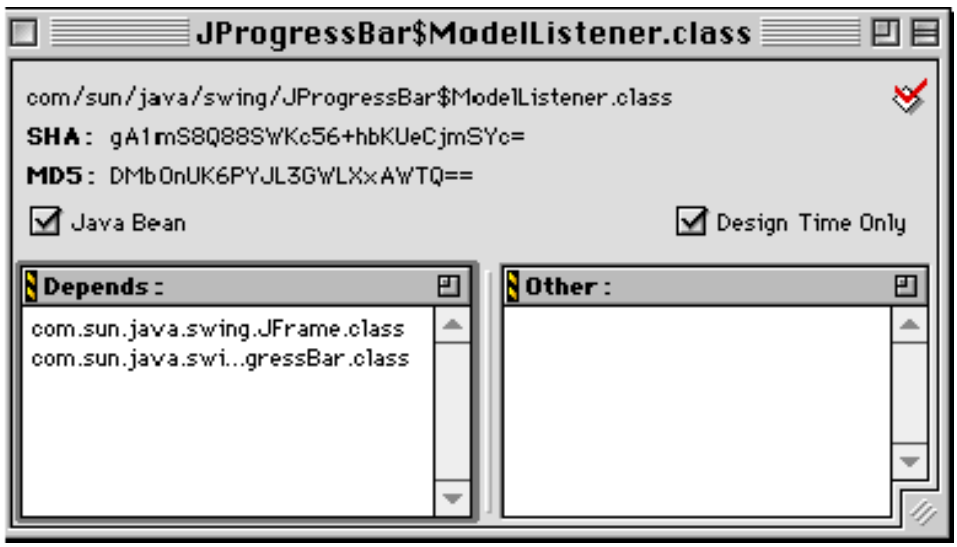
You can move a file from one archive to another. Open both archives in Class Wrangler, and drag the files from one window to another. Class Wrangler copies the files from the source archive into

the destination archive. The files are not removed from the source archive.

Editing Manifest Files

Class Wrangler also allows you to edit some manifest data for the currently selected item in the active archive. Choose **Edit Manifest** from the Package menu to edit the manifest data for the currently selected item.

Figure 8.5 Edit Manifest dialog



Manifest data is displayed for the item (Figure 8.5) for you to verify. This data reflects the state of the item in memory, not on disk. So the JavaBean item will display the state of that item as it would currently be saved to disk, not as it was originally.

Certain items, such as the name, **SHA**, and **MD5** hash codes, cannot be modified. The **Depends** list is a list of items. Any text can be dragged to this pane in order to add it to the list. The **Other** list holds all manifest tags that are not yet specifically parsed by Class

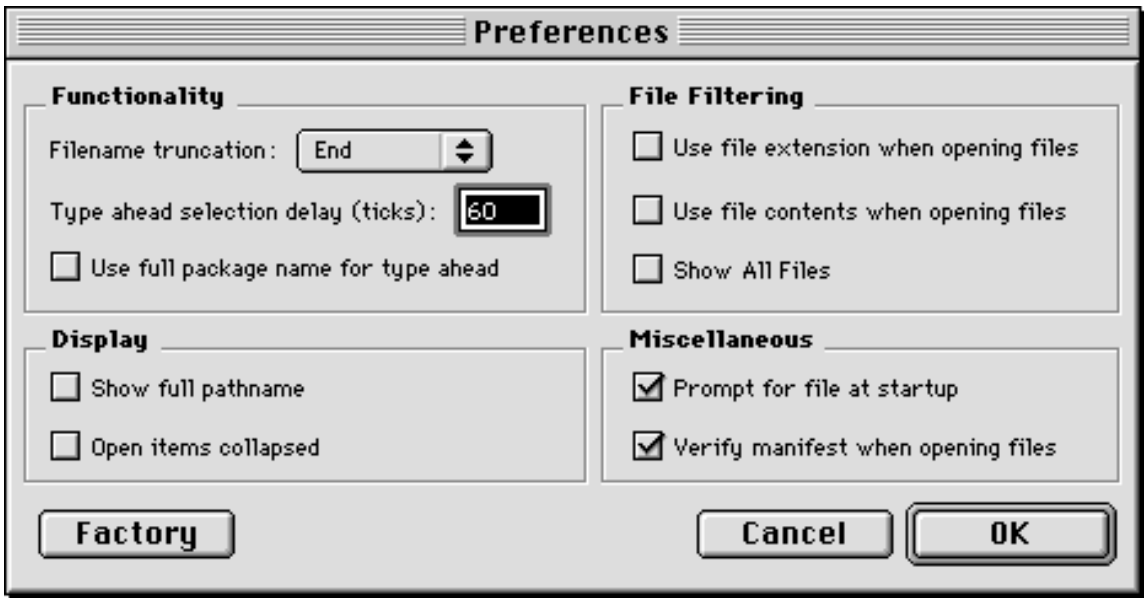
Wrangler. These could include new tags from Sun or special user-defined tags.

See also Sun's documentation for the complete explanation of manifest tags and the manifest file format.

Class Wrangler Preferences

This section discusses how to control the way Class Wrangler works for you. As with most programs, Class Wrangler has preferences you can set to control its operation. When you choose **Preferences** from the Edit menu, Class Wrangler displays a Preferences dialog box as shown in Figure 8.6.

Figure 8.6 The Class Wrangler Preferences dialog



Click the **Factory** button to restore the settings to program defaults.

The effect of these options is described in the following topics:

- Functionality Settings
- File Filtering Settings

- Display Settings
- Miscellaneous Settings

Functionality Settings

The first group of preference settings control how you interact with Class Wrangler.

Filename truncation

The **Filename truncation** popup menu operates similar to the same menu in the **Java Target** panel in the CodeWarrior IDE. On the Mac OS, file names cannot be greater than 31 characters long. If you extract a class file that is greater than 31 characters, Class Wrangler uses this setting to determine where the extra characters will be eliminated from. Options are: **Front**, **Middle**, and **End**.

If you choose **End**, Class Wrangler preserves any file name extension in the final file name.

Type ahead selection delay (ticks)

The amount of time, in ticks, Class Wrangler will wait before selecting the file represented by typing on the keyboard.

Use full package name for type ahead

Enable this option to force type ahead selection based on the full package name of the class. For example:

```
java/awt/Adjustable.class
```

File Filtering Settings

When you add or open a file using the standard file dialog box, Class Wrangler displays only zip, Jar, and class files in the file list. You control how Class Wrangler identifies the proper files with this group of options in the Preferences dialog.

If you use files created on Mac OS computers, turn off all options. When these options are off, Class Wrangler looks at a file's Mac OS

file type to determine whether a file is a zip archive, Jar file, or a class file. Class Wrangler assumes that class files have a file type of `'Clss'` or `'COåk'`. Zip archives have a file type of `'ZIP '` or `'ZipF'`. Jar files have a file type of `'ZIP '`.

If you use files created on other operating systems (such as Windows or UNIX), you'll need to turn on one of the options. When you transfer such a file to a Mac OS computer, the file does not have the proper Mac OS file type. Class Wrangler needs some other mechanism it can use to identify that the file is appropriate.

NOTE: These options don't affect drag and drop operations. To successfully drop a file, it must have the proper Mac OS file type.

Use File Extension when opening files

Will only show files with a `.zip`, `.jar`, or `.class` extension.

TIP: Use Internet Config to map common Java extensions, such as .zip and .jar to the appropriate Mac OS file types. See "Use Internet Config for File Mappings" for more information.

Use file contents when opening files

Will only show files based on their contents. This option is useful if a zip or Jar file does not have the appropriate extension.

Show All Files

Displays all files in the list regardless of file extension or content. This option overrides the other options, even if they are set.

TIP: Each of these options are also available to you in the Open dialog box.

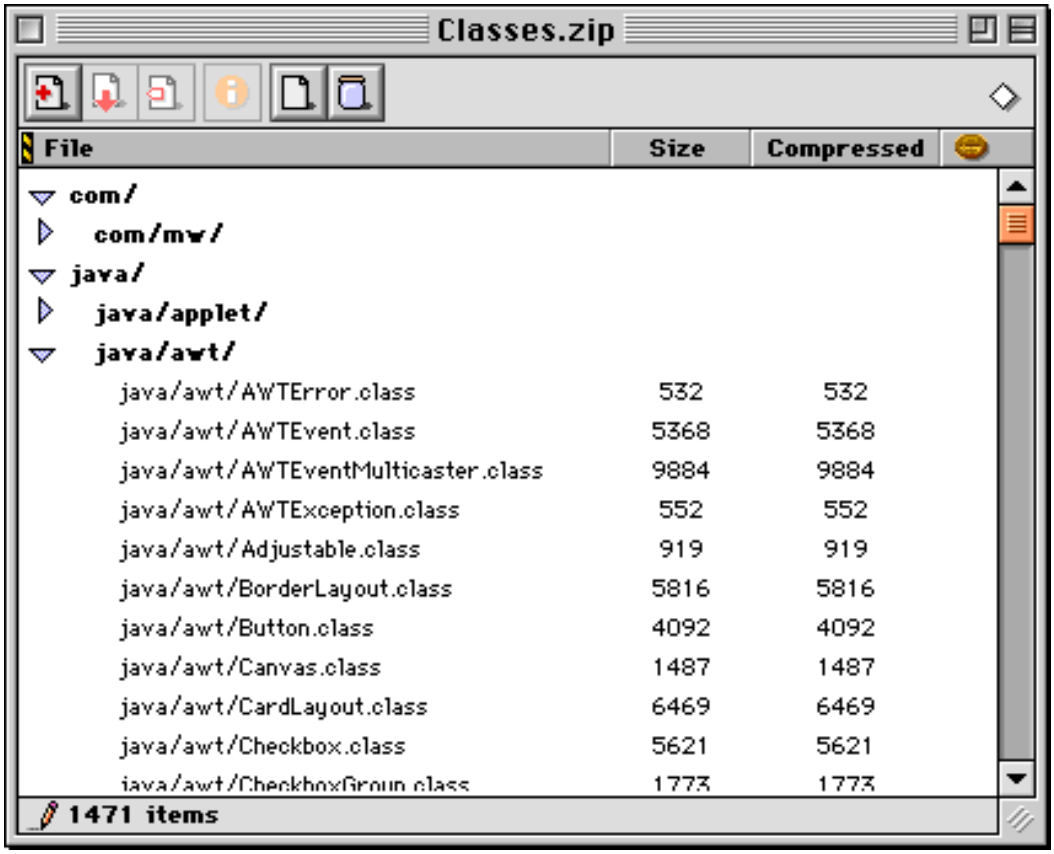
Display Settings

The display settings control how Class Wrangler displays a zip/Jar file.

Show full pathname

If the **Show full pathname** option is on in the Preferences dialog box, Class Wrangler shows the fully qualified name for every entry in its window, as shown Figure 8.7.

Figure 8.7 The Class Wrangler window with full pathnames



If this option is off, only the name of that entry's package or class file is shown.

Open items collapsed

If the **Open items collapsed** option is on in the Preferences dialog box, the program will always open archives with all hierarchical entries collapsed. Click the disclosure triangle to display the contents of a package.

Miscellaneous Settings

The last two settings tell Class Wrangler what to do at startup and when opening files.

Prompt for file at startup

If the **Prompt for file at startup** option is on, Class Wrangler will ask you to specify a file to open when you launch the program.

Verify manifest when opening files

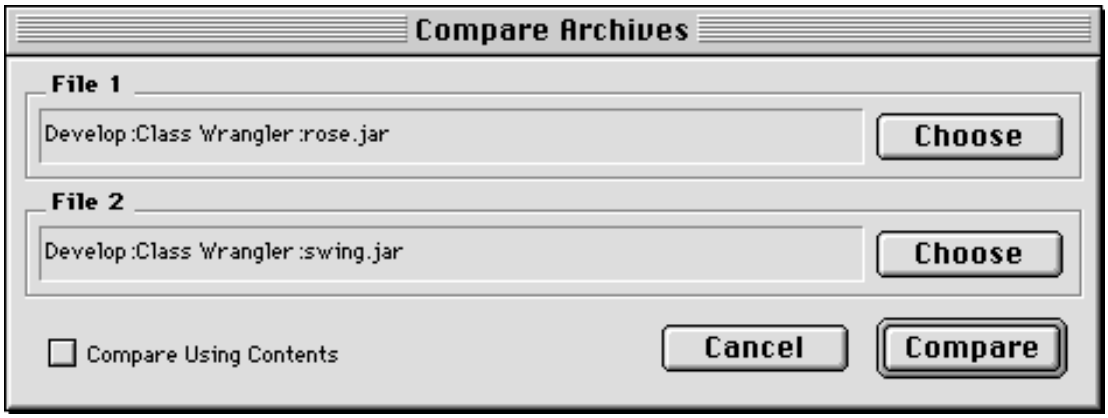
Verifies the manifest information, if present, of a Jar file as Class Wrangler opens the file. Jar files will take longer to open with this option enabled. If a particular entry that has a manifest entry doesn't verify, Class Wrangler draws it in red.

Comparing Archives

Class Wrangler allows you to compare two Jar files to determine their differences. You can compare archives on two levels. At the first level, comparison is based on the names and positions of items in the Jar file. The second level compares the Jar files by differences in the contents of each item.

To compare two Jar files, choose **Compare Files** from the File menu. The Compare Archives dialog, shown in Figure 8.8, is straight forward to use.

Figure 8.8 Compare archives dialog



Click the **Choose** button to pick the files you wish to compare. If you want to compare the files based on differences in the content of each item, click the **Compare Using Contents** checkbox. Click **Compare** to start the process.

NOTE: Comparing files requires a lot of RAM. This is especially true if you are comparing large Jar files. It may be necessary to increase the memory partition of Class Wrangler in some cases.

Once the compare process is completed, each archive is displayed in a separate window. Items that are unique to an archive are drawn in green. Items that are in both archives, but are different, are drawn in blue.

TIP: You'll only see items drawn in blue with the **Compare Using Contents** option enabled.

9

Metrowerks Java

This chapter explains the Metrowerks Java application. This is a combination Java applet runner and virtual machine for Mac OS. The information in this chapter details the inner workings of this tool, and its various elements.

To develop most applications or applets, you won't need the information here. "Creating Java Projects" should direct you to everything you need to know.

The topics discussed include:

- What Is Metrowerks Java?
- Running Java Applications
- Running Applets with Metrowerks Java
- Disassembling Classes
- Verifying Classes
- Compiling Java Code with the Sun Compiler
- Generating C Files for Native Methods
- Metrowerks Java Preferences

What Is Metrowerks Java?

Metrowerks Java is a single application that contains a set of six utilities for Java development, including a Java interpreter and an applet viewer. It also contains Mac OS versions of some command-line programs that Sun includes in its Java Developer's Kit (JDK). All six utilities are found in the Windows menu.

Metrowerks Java also contains interfaces to use any Java VM that conforms tot he JManager interface (currently only the Metrowerks VM, Metrowerks JIT VM, and Apple's MRJ).

This section gives you an overview of what these utilities are and what they do. Topics covered include:

- Utilities for Running Applications and Applets
- Utilities for Advanced Operations
- The Java Output window
- Classpaths Dialog

Utilities for Running Applications and Applets

Normally you run Java projects directly from the IDE, which takes advantage of the services of Metrowerks Java. However, you can also run them directly from Metrowerks Java. These are utilities you'll use frequently, either directly or indirectly.

- To run an application, choose **Application** or **Interpreter** .
- To run an applet, choose **AppletViewer** .

Interpreter

The Interpreter utility is a Mac OS version of Javai. Generally, you'll run a Java application by dragging and dropping it on Metrowerks Java. However, you can specify command-line arguments in the Interpreter window, shown in Figure 9.1. For more information, see "Running Java Applications."

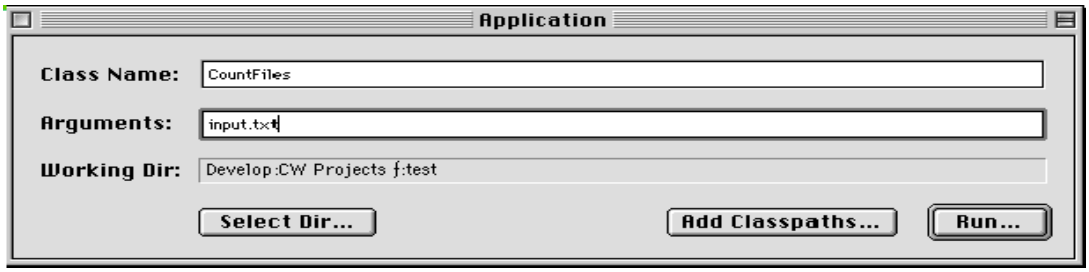
Figure 9.1 Interpreter

NOTE: A Jar file can contain more than one application, so running it may not have the effect you intend. Examine the Jar file with Class Wrangler first. See "Using Class Wrangler" for more information.

Application

The Application utility (Figure 9.2) is similar to the interpreter, but does not allow the extra javai command switches.

Figure 9.2 Application



AppletViewer

The AppletViewer window runs your Java applets. Generally, you'll run an applet directly from the IDE. However you can run any applet by specifying its address in the AppletViewer window, shown in Figure 9.3 or drag and drop `.html` files onto Metrowerks Java. For more information, see "Running Applets with Metrowerks Java."

Figure 9.3 AppletViewer

WARNING! Do not use the AppletViewer to run untrusted applets. It is designed for running and debugging your own applets. Although it includes the normal security manager package for Java applets, it is not intended to be a replacement for a fully-secure and validated Java-enabled HTML browser.

Utilities for Advanced Operations

These are utilities you'll use less frequently. These utilities can only be used with the Metrowerks VM. They cannot be used with the Metrowerks JIT VM or MRJ.

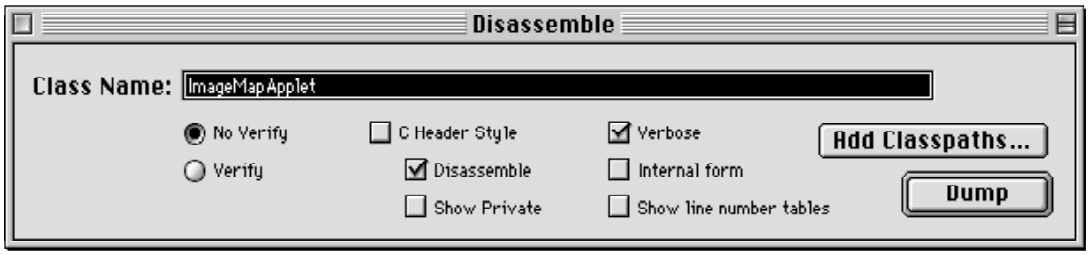
- To disassemble and verify programs, choose **Disassembler** .
- To compile a file with the Sun Java compiler, choose **Compiler**.
- To implement a class in C, choose **Native Interface** .

Disassembler

The Disassembler utility (Figure 9.4) performs two different functions: bytecode verification, and disassembly. If the **Verify** option is on, it runs the bytecode verifier over a class without running the class's code. If the **No Verify** option is on, it disassembles a class to produce a listing of the Java Virtual Machine instructions that implement the class. For more information, see "Verifying Classes" and "Disassembling Classes."

The Disassembler is only available when using the Metrowerks VM.

Figure 9.4 Disassembler window

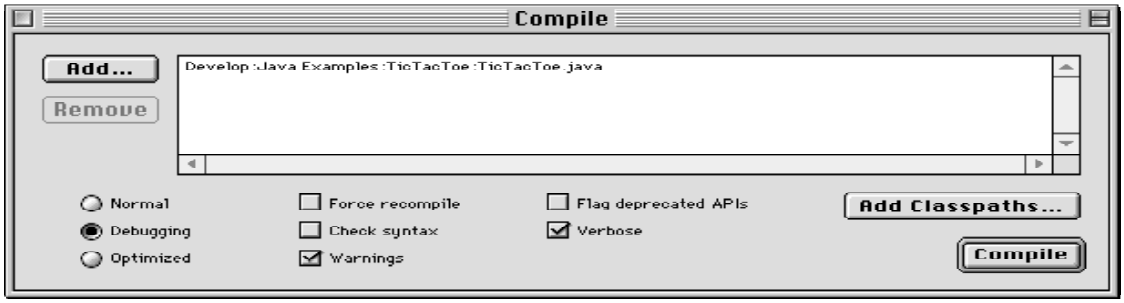


Compiler

The compiler (Figure 9.5) lets you compile a Java file with the Sun Java compiler. Ordinarily, you will use the Metrowerks Java compiler that runs under CodeWarrior, but the Sun Java compiler is available. For more information, see "Compiling Java Code with the Sun Compiler."

The Compiler is only available when using the Metrowerks VM.

Figure 9.5 Compiler window

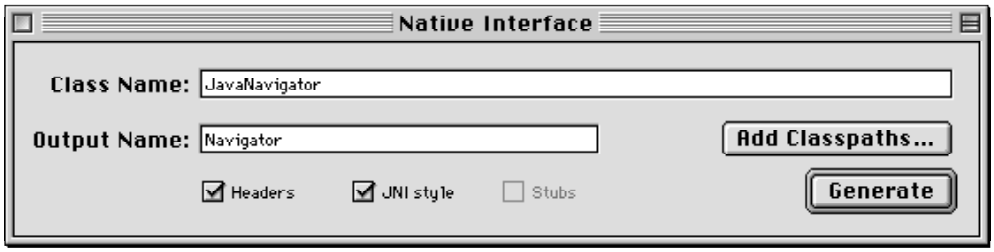


Native Interface

The Native Interface utility (Figure 9.6) creates C files that let you implement in C any classes you declared as `native`. For more information, see "Generating C Files for Native Methods."

The Native Interface is only available when using the Metrowerks VM.

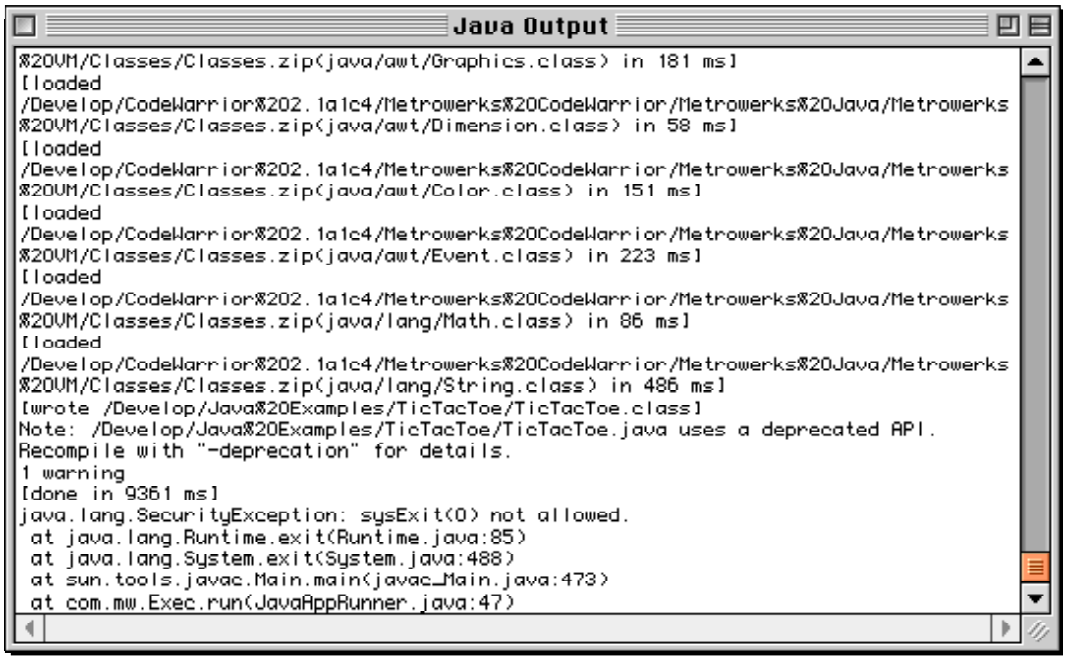
Figure 9.6 Native Interface window



The Java Output window

The Java Output window (Figure 9.7) is where Metrowerks Java displays `System.out`, `System.err`, and its own status messages. To view this information, choose **Output Window** from the Windows menu.

Figure 9.7 The Java Output window

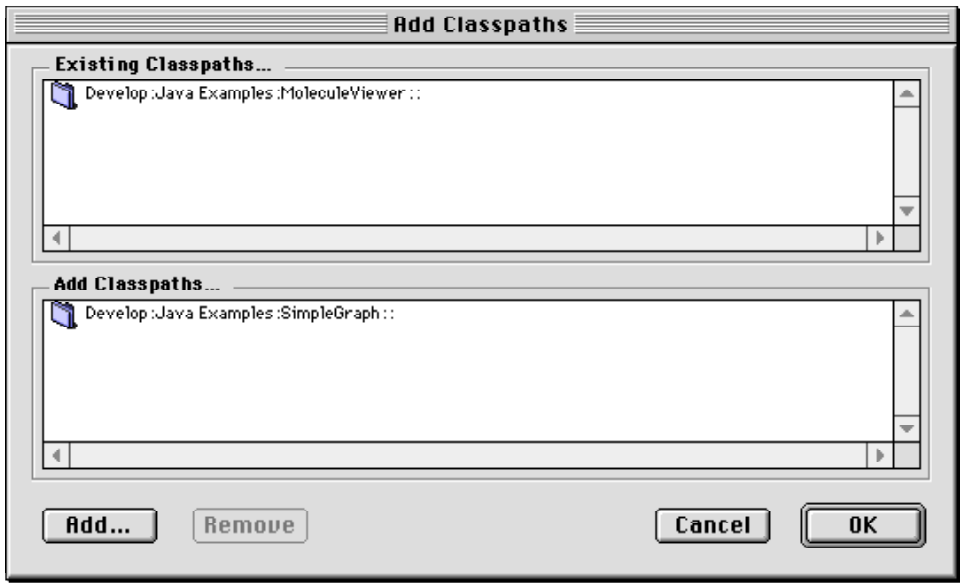


Classpaths Dialog

The Classpaths dialog box (Figure 9.8) allows you to add, remove, and view class paths for the active session. You can access this dialog by choosing **Classpaths** from the Windows menu, or by clicking the **Add Classpaths** button in many of Metrowerks Java utility windows.

NOTE: You cannot remove classpaths from the **Existing Classpaths** field by selecting them and clicking the **Remove** button. Instead, choose **Reload VM** from the Edit menu. This closes down the VM and Reloads it, clearing the classpaths in the process.

Figure 9.8 Classpaths dialog



Running Java Applications

You can run Java applications from within Metrowerks Java with the Interpreter or Application utilities. Depending of the type of application you created, there are different ways to run it. You set the type of application with the **Type** menu in the **Java Target** panel in the CodeWarrior IDE.

NOTE: If the program is an applet, see "Running Applets with Metrowerks Java."

- If the project **Type** is **Jar File** or **Class Folder** , you can choose between "Using Interpreter to Run an Application" or "Using the Application Utility."
- If the project **Type** is **Application** , drag and drop the file onto Metrowerks Java. Metrowerks Java loads it and automatically runs the `main()` method of the class you entered in the **File Name** field in the **Java Target** panel in the CodeWarrior IDE. If you want to run the application again,

you can choose between "Using Interpreter to Run an Application" or "Using the Application Utility."

The rest of this section covers the following topics:

- Using Interpreter to Run an Application
- Specifying Java Interpreter Options
- Using the Application Utility

Using Interpreter to Run an Application

You can use the Interpreter to run any type of Java application. Simply follow these steps.

1. **Add a path to your application**

   Choose **Classpaths** from the Windows menu. Click the **Add** button in the Add Classpaths dialog box. Navigate to the folder containing your application files, click **Select**.

2. **Choose Interpreter from the Windows menu.**

3. **Enter the class's name or a command line in the text box.**

   Enter the class name and any command line arguments as shown in Figure 9.9.

Figure 9.9 The interpreter window

If you've run the program since you launched Metrowerks Java, enter the name of the class whose `main` method you want to run, and any arguments you'd like to pass to it, like this:

```
CountFile "Develop:count.txt"
```

If you haven't run the program already, you also need to specify it's location with the `-classpath` option, like this:

```
-classpath "Develop:Java" CountFile "Develop:count.txt"
```

There are other options that let you control how Metrowerks Java runs your application. For example, you can give it more memory or turn off the garbage collector. For more information, see "Specifying Java Interpreter Options."

4. **Click the Execute button.**

   Metrowerks Java runs the application.

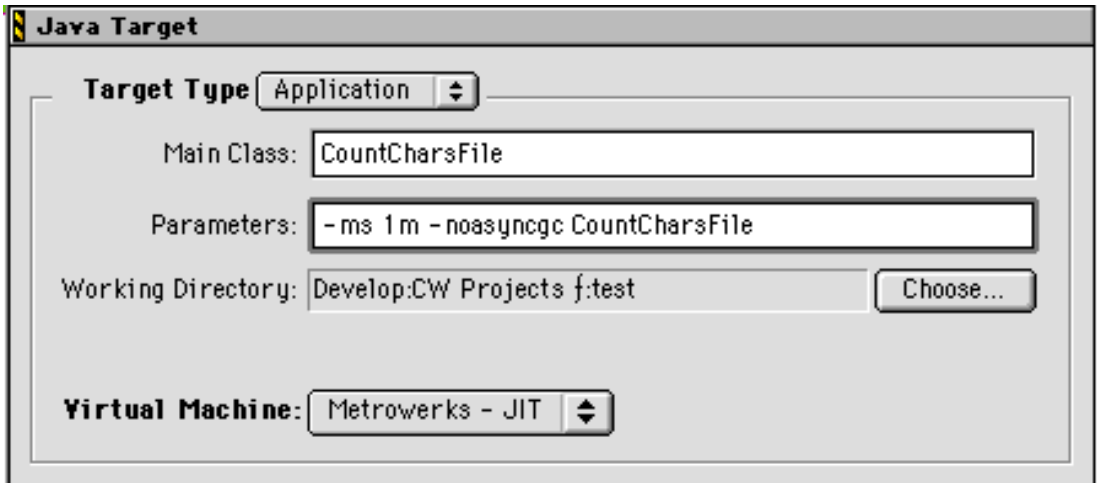
Specifying Java Interpreter Options

Metrowerks Java lets you specify how it runs your application. For example, you can give it more memory or turn off the garbage collector. You specify these options in either:

- The text box in the Interpreter window of the Metrowerks Java application.
- The Parameters field in the **Java Target** panel of the CodeWarrior IDE.

Figure 9.10 The Interpreter window, with a command line

Figure 9.11 The Java Target panel, with a command line



Just enter an expression of this form:

[*interpreter-options*] *class-name* [*program-arguments*]

The *interpreter-options* let you control how Metrowerks Java runs your application. The *class-name* is the name of the class whose `main` method you want Metrowerks Java to run. The *program-arguments* are the arguments that Metrowerks Java passes to the `main()` method using the `args` array.

The rest of this section describes the options you can set. Topics discusses are:

- Setting the pathnames
- Setting memory size
- Allowing garbage collection
- Printing garbage collection messages
- Printing class-loading messages
- Verifying Java bytecodes
- Getting help
- Printing options

Setting the pathnames

`-classpath` *pathname*`[,` *pathname*`]`
`-working` *pathname*

These options let you specify where the application searches for files. Both -classpath and -working are passed in using Mac OS conventions (using a colon ":" for the directory name divider) but Java code uses Unix conventions (a forward slash "/" is used as the directory path divider).

To specify where the interpreter searches for classes, use the `-classpath` option. Use the pathnames of the directories or Jar files for the specified class and all other classes that it uses. Separate multiple items with a semicolon.

To specify where the application searches for files when it's running, use the `-working` option. In file names, `./` stands for the working directory. For example if you specify `-working"Jeff:temp:"` in the command line, and you use the file name `"./test"` in your Java code, the application uses the file `"Jeff:temp:test"`.

NOTE: If you use both the `-working` and the `-classpath` switches, make sure that the `-working` switch comes first. The `-working` switch removes any items you added with the `-classpath` switch. This is because `-working` may have relative pathnames (relative to the working directory).

Setting memory size

`-ms` *initsize*`[k|m]`
`-mx` *maxsize*`[k|m]`
`-oss` *stacksize*`[k|m]`
`-ss` *stacksize*`[k|m]`

These options let you specify how much memory the Java interpreter uses. You can specify the amount of memory in bytes, kilobytes, or megabytes. To specify megabytes, use `m` after the number. To specify kilobytes, use `k` after the number. To specify bytes, don't use any letter after the number.

- To set the initial size of the heap, use `-ms`. The default size is 512K.

- To set the maximum size of the heap, use `-mx`. The default size is 8 MB.
- To set the size of each thread's Java code stack, use `-oss`. The default size is 400K.
- To set the size of each thread's native code stack, use `-ss`. The default size is 64K.

Allowing garbage collection

```
-noasyncgc
-asyncgc
```

To perform garbage collection in its own thread, use `-asyncgc`. If you use `-noasyncgc`, the Java Interpreter performs garbage collection only when it runs out of memory or when the program explicitly calls the garbage collector. The default is `-asyncgc`.

Printing garbage collection messages

```
-verbosegc
-noverbosegc
```

To print a message whenever the garbage collector frees memory, use `-verbosegc`. To perform garbage collection without printing a message, use `-noverbosegc`. The default is `-noverbosegc`.

Printing class-loading messages

```
-v   | -verbose
-nv  | -noverbose
```

To print a message whenever the Java Interpreter loads a class, use `-verbose` or `-v`. To load classes without printing a message, use `-noverbose` or `-nv`. The default is `-noverbose` (or `-nv`).

Using the accelerated interpreter

```
-accel
-noaccel
```

Metrowerks Java contains an accelerated interpreter that's optimized for all Mac OS computers. With the accelerated interpreter, your code runs 20-50% faster than with the standard Sun interpreter.

To use the accelerated interpreter, use the `-accel` option. To use the standard Sun interpreter, use the `-noaccel` option. By default the `-accel` option is on.

Verifying Java bytecodes

```
-verify
-verifyremote
-noverify
```

These options let you choose whether the Java interpreter verifies Java code before it runs the code. Verification ensures that the bytecode you're running is valid Java Virtual Machine code, and does not perform illegal operations, such as forging pointers or crashing the Java interpreter, which could give it access to your computer. It's especially useful when you're running code you're not familiar with, including code from over the Internet.

To verify all Java code, use `-verify`. To verify only Java code which comes from another computer (such as over the Internet), use `-verifyremote`. To verify no Java code, use `-noverify`. The default is `-verifyremote`.

Getting help

```
-help
```

To print short descriptions of the options you can use in the Java Interpreter, use the `-help` option. The Java Interpreter prints one-line descriptions of all the options but does not run your program.

Printing options

```
-stat
```

To print a list of the current option settings for this command, use the `-stat` option. The Java Interpreter prints a summary of the option settings and then runs your program. All memory sizes are printed in bytes.

`-stat` can also be used by itself to get a useful snapshot of the VM state.

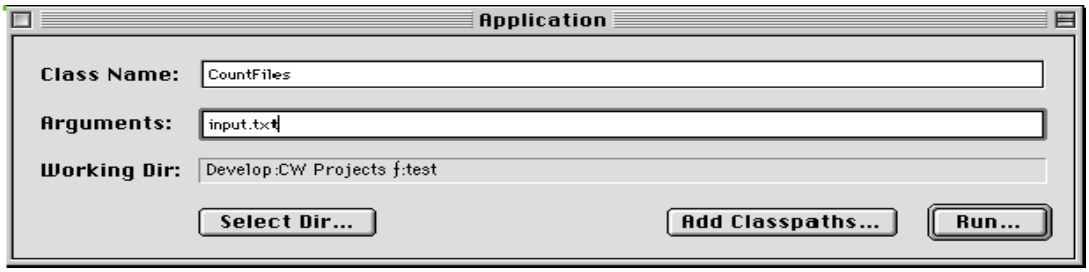
Using the Application Utility

The application utility is very similar to the Interpreter utility, but you cannot specify any command line switches, like `-classpath`.

If you open a zip archive or Jar file with the **Open** command, a classpath to the archive is set and the application window is automatically opened for you.

The application utility needs to know the main class name in order to run your Java application. Type the your class name with the `main()` method in the **Class Name** field and any arguments in the **Arguments** field. Click **Run** to execute the application.

Figure 9.12 Application utility



If you do not open a zip or Jar file and decide to use the Application utility manually, do not forget to add a path to your compiled application with the **Add Classpaths** button.

Running Applets with Metrowerks Java

The AppletViewer runs Java applets, small programs that are specially designed to run in an HTML browser. You can run an applet directly from Metrowerks Java in two different ways. You can open a local HTML file with the **Open** command, or specify a URL in the AppletViewer window.

This section contains:

- Running a Local Applet with MW Java

- Running a Remote Applet with MW Java
- Changing the Applet's Network Security

Running a Local Applet with MW Java

Choose **Open** from the Metrowerks Java File menu. In the standard file dialog box, choose any HTML file that uses the applet.

TIP: Alternatively, you can open the AppletViewer window and click the **Locate** button to open a local HTML file.

Metrowerks Java searches for the `<applet>` and `</applet>` tags and immediately executes the specified applet with the specified parameters in its own window.

If the Metrowerks Standard or JIT MV is the chosen VM in the **Program** preference panel, your HTML file may include multiple zip archives in one applet tag like this:

```
<applet archive="main.zip, foobar.zip" code="Main.class" width=200
height=200> </applet>
```

Running a Remote Applet with MW Java

You can also use Metrowerks Java to locate, download, and launch an applet from the Internet (or any remote location on a network.) To accomplish this task, simply follow these steps.

1. **Open AppletViewer.**

   Choose **AppletViewer** from the Windows menu.

2. **Specify the URL.**

   In the URL field, enter the address of an HTML page that uses the applet. The window should look like the one below:

Figure 9.13 The AppletViewer window, with a URL

3. **Click the Run button**

   Metrowerks Java loads the file from the Internet, searches for the `<applet>` and `</applet>` tags, and immediately executes the specified applet with the specified parameters in its own window.

WARNING! Do not use the AppletViewer to run untrusted applets. It is designed for running and debugging your own applets. Although it includes the normal security manager package for Java applets, it is not intended to be a replacement for a fully-secure and validated Java-enabled Web browser.

TIP: If you use the Metrowerks VM or JIT, you can also select teh Sun Applet Viewer. See "Use Sun AppletViewer."

Changing the Applet's Network Security

If you're running an applet that you're not familiar with, you may want to change it's network security to protect your computer from harm. To change the network security, go to the **VM** panel in the Preferences dialog box. For more information, see "Metrowerks Java Preferences" and "VM."

Disassembling Classes

A disassembly of a class or Jar file produces a listing of the Java Virtual Machine instructions that implement the class.

You can view a disassembly of a file in the IDE, the debugger, or using Metrowerks Java and the Disassembler utility.

In the CodeWarrior IDE, you can choose **Disassemble** from the Project menu after selecting a Java source file. For more information on disassembling source files, see the *IDE User Guide*.

In the Metrowerks Debugger, you can use the pop-up menu at the bottom of any Source or Program window. For more information on viewing disassembly in the debugger, see "Viewing the Java Virtual Machine Disassembly."

This section discusses how to accomplish the same goal using Metrowerks Java. The topics in this section are:

- Disassembling a Class
- Setting the Disassembler Options

The Disassembler utility can also verify the bytecodes of a class. For more information, see "Verifying Classes."

Disassembling a Class

To disassemble a class using the Disassembler utility of Metrowerks Java, follow these steps.

1. **Make sure the Jar or Class file is in the class path or working directory.**

   If you want to disassemble a class file, choose **Classpaths** from the Windows menu. Click the **Add** button in the **Add Classpaths** dialog box. Navigate to the folder containing your files, click **Select**.

   If you want to disassemble an application in a Jar file, make sure you select the Jar file when you add the classpath or the disassembler will not disassemble the code.

2. **Open the disassembler.**

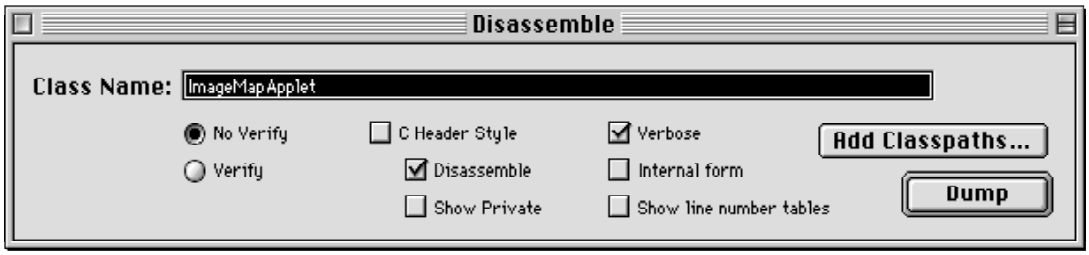
   Choose **Disassembler** from the Windows menu.

3. **Enter the class name to disassemble**

   Enter the name of the class you want to disassemble (Figure 9.14). You cannot add any extensions, such as `.class`, `.jar`, or `.zip`.

However, you can use a fully qualified name, such as `java.lang.Thread`.

Figure 9.14 Disassembler window



4. **Turn on the No Verify option.**

   When you turn on the **No Verify** option, Metrowerks Java disassembles the file. If you turn on the **Verify** option, Metrowerks Java runs the bytecode verifier over the file, but does not print a disassembly of it.

5. **Set other options.**

   With no other options on, the Java Disassembler prints only a summary of the class's methods. To print a disassembly as well, turn on the **Disassemble** option. To print debugging tables, turn on the **Show line number tables** option.

   For more information on these options, see "Setting the Disassembler Options."

6. **Click the Dump button.**

   Metrowerks Java prints a summary of the methods and optionally prints a disassembly and debugging tables.

Setting the Disassembler Options

The Java Disassembler prints a summary of the class's methods and optionally prints a disassembly of the class and two debugging tables.

The summary of the class's methods contains the declarations of the `public` methods, constructors (even implicit constructors), and static initializers of the classes specified in the **Class Names** field.

The **Show Private** , **C Header Style** , and **Verbose** options affect how the Java Disassembler displays these declarations. For more information, see "Formatting the method summary."

To print a disassembly of the Java Virtual Machine instructions or debugging tables, use the **Disassemble** and **Show Line Number tables** options. For more information, see "Printing the disassembly" and "Printing line numbers and local variable tables."

Formatting the method summary

To print the declarations of the private and protected methods, turn on the **Show Private** option. If this option is off, the Disassembler prints only the public methods.

To print the declarations of the class's methods using C syntax instead of Java syntax, turn on the **C Header Style** option. Turning this option on greys out the **Disassemble** and **Show Private** options.

To print how much space each method needs for local variables, arguments, and stack, turn on the **Verbose** option.

Printing the disassembly

To print the Java Virtual Machine instructions for each of the methods in each of the classes specified in the **Class Names** field, turn on the **Disassemble** option.

This option disassembles all methods, including `protected` and `private` methods, regardless of the setting of the **Show Private** option.

Printing line numbers and local variable tables

To print line numbers and local variable tables, turn on the **Show Line Number Tables** option. The Java Disassembler prints two types of tables. One displays which line numbers in the Java source file correspond to which disassembled Java Virtual Machine instructions. The other displays detailed information about the local variables in each method in the class.

Note that this information may not be available, depending on which options you set. If you compiled the file with the Metrowerks Java compiler, both line numbers and local variable tables are always available. If you compiled the file with the Sun Java compiler, local variable tables are available if you compiled the file with the **Debugging** option turned on, and the line number information is available if you compiled the file with the **Debugging** option turned on or with the **Optimized** option turned off.

Verifying Classes

One function of the Disassembler utility is to run the bytecode verifier over a class. Verification ensures that the bytecode is valid Java Virtual Machine code, and does not perform illegal operations, such as forging pointers or crashing the Java interpreter, which could give it access to your computer. It's especially useful when you're running code you're not familiar with, including code from over the Internet.

TIP: The Disassembler utility can also print a disassembly of a class. For more information, see "Disassembling Classes."

To use the Disassembler to verify bytecode, follow these steps.

1. **Make sure the program is in the class path or working directory.**

   If you want to verify a class file, choose **Classpaths** from the Windows menu. Click the **Add** button in the Add Classpaths dialog box. Navigate to your the folder your application files, click **Select**.

   If you want to verify an application in a zip or Jar file, make sure you select the zip/Jar file when you add the classpath or the disassembler will not verify the code.

2. **Open the Disassembler.**

   Choose **Disassembler** from the Windows menu.

3. **Turn on the Verify option.**

   When you turn on the **Verify** option, Metrowerks Java runs the bytecode verifier over the class. If you turn on the **No Verify** option, Metrowerks Java disassembles the class instead of verifying it.

4. **Click the Dump button.**

   Metrowerks Java runs the bytecode verifier over the class.

Compiling Java Code with the Sun Compiler

The Compiler utility lets you use the Sun Microsystems Java compiler. Most of the time you'll use the Metrowerks Java compiler that works under the CodeWarrior IDE. However you may want to use the Sun Java compiler if you need to use the reference compiler.

This section explains how in these topics:

- Compiling a File with the Sun Java Compiler
- Setting Options for the Sun Java Compiler

Compiling a File with the Sun Java Compiler

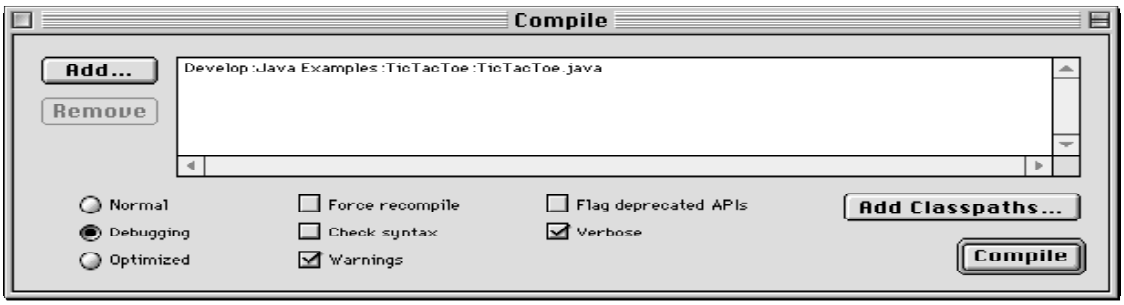
1. **Open the Compiler window.**

   Choose Compiler from the Windows menu.

2. **Add your java files.**

   Click the **Add** button to add your java files to the project. You can compile multiple files by clicking the **Add** button for each file before you compile.

Figure 9.15 The Compile window



3. **Set your options.**

   For more information on these options, see "Setting Options for the Sun Java Compiler."

4. **Click Compile.**

   The Sun Java compiler compiles the file and any changed files that it imports. It produces a class file for each class it compiles.

NOTE: You may need to add classpaths if your Java code references classes that are not in the standard library, or are not in the same directory as your code.

Setting Options for the Sun Java Compiler

This section describes all the options you can set for the Sun Java Compiler. You set these options in the Compiler window. The items in the window are:

- Check syntax
- Normal, Debugging or Optimized
- Verbose
- Warnings
- Force recompile
- Add button

Check syntax

To check the syntax of the Java file without compiling it, turn on the **Check Syntax** option. The Sun Java Compiler compiles the file and prints any appropriate warning and error messages, but it does not create a new class file.

Normal, Debugging or Optimized

You may choose only one of these options.

To include debugging information in the compiled code, turn on the **Debugging** option. The Sun Java Compiler adds line numbers and local variable information to the output class files.

To optimize the compiled code, turn on the **Optimized** option. The Sun Java Compiler does not include line numbers or local variable information in the compiled code and inlines methods declared `static`, `final`, and `private`. Turning this option on may make your code faster, but it may also make your class files larger and you will not be able to debug them.

To create code without debugging information and that is not optimized, turn on the **Normal** option. The Sun Java Compiler does not include line numbers and doesn't inline any methods, but does include local variable information.

Verbose

To have the Sun Java Compiler print status messages as it compiles the file, turn on the **Verbose** option.

Warnings

To have the Sun Java Compiler print warning messages in addition to error messages, turn on the **Warnings** option. Note that if the **Warnings** option is off, the compiler still prints error messages.

Force recompile

To recompile every file that this file imports, turn on the **Force Recompile** option. If this option is off, the Sun Java Compiler recom-

piles an imported file only if it's been changed since it was last compiled.

Note if the Sun Java Compiler cannot find the source file for an imported file, it does not recompile the file and does not raise an error. This keeps the compiler from re-compiling the AWT, for example, each time you compile a file with the **Force Recompile** option on.

Add button

To compile a different file, click the **Add** button and select the file from the dialog that appears. The name of the file appears to the right of the **Add** button.

Generating C Files for Native Methods

Some system dependent functionality cannot be implemented in Java and must be written using native code. This is the primary purpose for native code.

The Native Interface utility lets you implement `native` class methods in C. Implementing some methods in C may speed up your Java application, but will cause problems when you port the application to other platforms, since you must recompile the C methods for each platform. Also note that you cannot use C methods in an applet, since you do not know what platform they may run on.

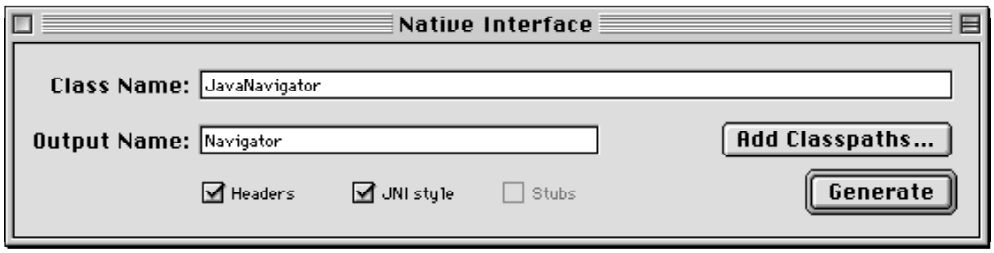
In your Java file, declare the methods that you want to implement in C as `native`. Then, use the Native Interface utility to create a C header and source file for the class:

- The header file declares a C struct that contains:
  - a field for each instance variable in the class, and
  - a function for each method in the class that you declared `native`.
- The C source file contains stub procedures that are necessary for linking the implementations of the native methods into your Java program. Do not enter the implementations of the native methods in this source file. You must create your own source file for them.

NOTE: You cannot use native methods with Applets due to restrictions about accessing native code in order to protect your system.

The **Native Interface** window is shown in Figure 9.16.

Figure 9.16 The Native Interface window



In the **Class Names** field, enter the names of the classes that you want to create native methods for, separated by spaces. In the **Output Name** field, enter the name that the Native Interface utility will use for the source and header files. The Native Interface utility will add a suffix to whatever you enter here: `.c` for the source file and `.h` for the header file.

Click **Add Classpaths** to make sure the program is in the class path. When you're using an application make sure you select the zip/Jar file when adding the class path.

To create a header file for the specified classes, turn on the **Header** option. To create a source file for the specified classes, turn on the **Stubs** option.

To create JNI (Java Native Interface) headers, click the **JNI style** option.

To generate the C files, press the **Generate** button.

Metrowerks Java Preferences

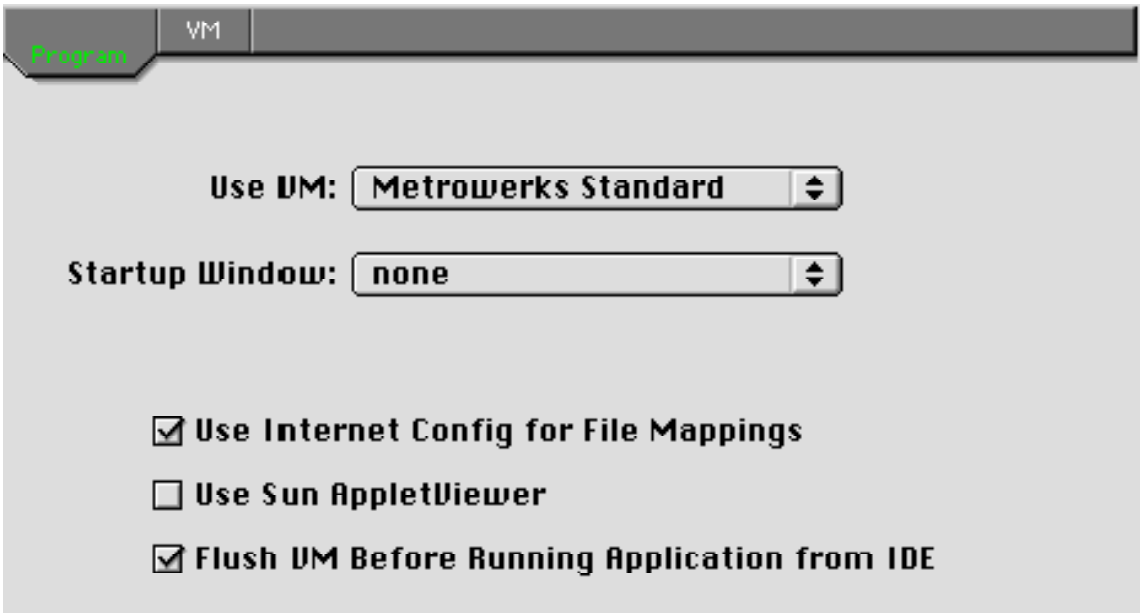
This section describes the two preference panels in Metrowerks Java:

- Program
- VM

Program

The program preferences, shown in Figure 9.17, control how Metrowerks Java operates.

Figure 9.17 Program preferences



Use VM

Choose a VM to use when running your Java code. Options are **Metrowerks Standard** , **Metrowerks JIT** , and **Apple MRJ** .

NOTE: The Native Interface, Disassembler, and Interpreter utilities are not available with the Apple MRJ VM.

Metrowerks Java allows you to change the VM at any time. However, it may take a short time to load the new VM after you click **Save**.

If a particular VM is not installed, the menu option is dimmed.

Startup Window

Choose a window to be displayed automatically, every time you launch Metrowerks Java. Options are: **AppletViewer** , **Application**, **Compiler** , and **none**.

Use Internet Config for File Mappings

Use this to allow Metrowerks Java to open files with other extensions as specified in Internet Config.

Table 9.1 lists some extensions and file types you should have mapped in Internet Config. Note where extra spaces are included in the file type.

Table 9.1 Internet Config file mappings

File extension	File type
.java, .jav, .JAVA, .JaV, .JAV	'TEXT'
.zip	'ZIP ', 'zip ', or 'ZipF'
.jar	'ZIP '
.class	'Clss' or 'COåk'
.HTML, .html, .htm, .HTM	'TEXT'

Use Sun AppletViewer

This option uses Sun's applet viewer when you choose Applet-Viewer from the Windows menu. This option is not available if Apple MRJ is the chosen VM.

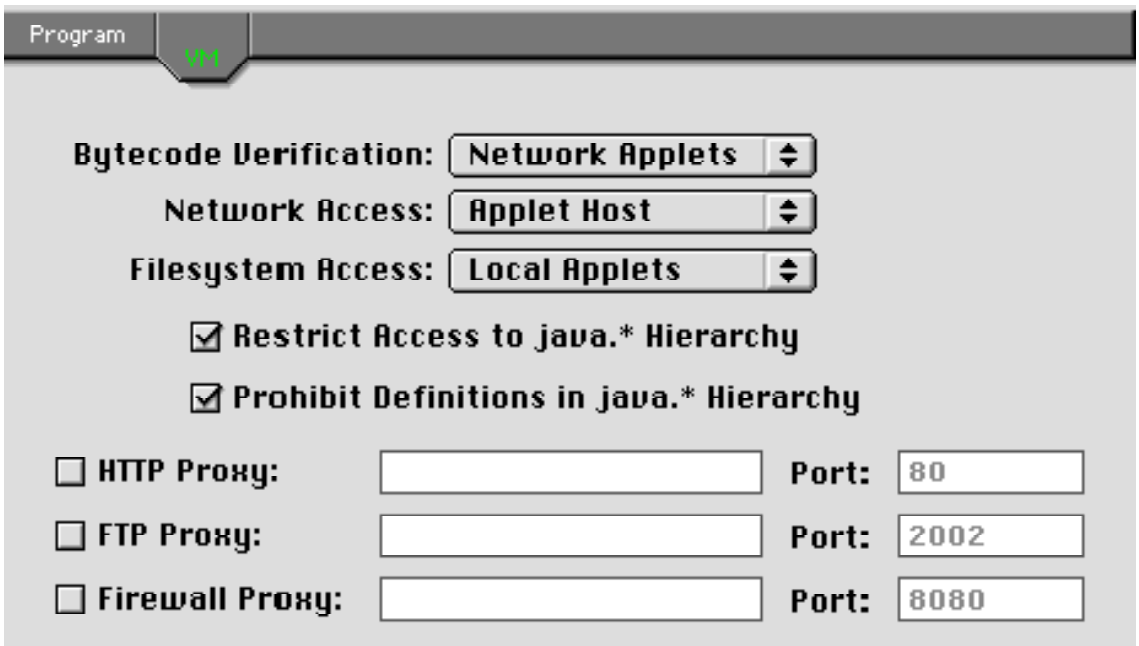
Flush VM Before Running Application from IDE

Flushes VM when the IDE runs a Java application using Metrowerks Java as the applet viewer. This preference is enabled by default.

VM

The **VM** preference panel (Figure 9.18) controls how the VM interacts with your Java applets and applications.

Figure 9.18 VM preferences



The first five options control the security permissions java code has while running on your computer.

Bytecode Verification

Controls which applets will be verified by Java bytecodes.

- **None** performs no bytecode verification (no security).
- **Network Applets** preforms bytecode verification on applets you access across a network, like the Internet (moderate security).
- **All Applets** performs bytecode verification on all applets (maximum security).

Network Access

Whether to allow the applet to have network access:

- **None** gives it no network access.
- **Applet Host** gives the applet access only to the computer it is from.
- **Unrestricted** gives the applet unlimited network access.

Filesystem Access

Whether to allow the applet to have access to your local file system:

- **None** gives no file system access.
- **Local Applets** gives only local applets access.
- **Unrestricted** gives all applets unlimited.

The last few options control how your applet accesses the Internet. Table 9.2 describes the different Proxy options.

Table 9.2 Proxy options

This field...	Specifies this...
HTTP Proxy Server	The firewall's HTTP proxy server.
HTTP Proxy Port	The port on the firewall's HTTP proxy server.

This field...	Specifies this...
Firewall Proxy Server	The firewall's proxy server for all other connections.
Firewall Proxy Port	The port on the firewall's proxy server for all other connections.

10

Troubleshooting on Java

This section is designed to give you a quick reference point for common problems with their solutions when using CodeWarrior on Java. This should be the first place you look before contacting technical support.

Troubleshooting questions and answers cover the following main categories:

- Programming Problems
- Conversion Problems

Programming Problems

This section deals with problems encountered writing Java code using the CodeWarrior development environment.

There are no programming problems to report at this time. If you find you are having problems in this area, please send a bug report to Metrowerks support, `support@metrowerks.com`, and that information will be added to this manual in the next release.

Conversion Problems

This section deals mainly with problems encountered converting older CodeWarrior Java projects to the latest release.

Can't Convert Older Droplet Projects

Problem

The Droplet project type was used in previous versions of CodeWarrior to make a "pseudo" stand alone applet. Droplet is no longer supported by CodeWarrior.

Solution

While the Droplet type is no longer supported, you can still make a Stand Alone Java application with CodeWarrior. The CodeWarrior project converter does not convert your project to the new type for you. The best method is to use the StandAlone Applet stationery and create a new project. Then add your Java source files to this project.

See "StandAlone Applets" for more information.

A

StandAlone Applets

You can use CodeWarrior to create double-clickable Mac OS Java applications. This release of CodeWarrior introduces a new method for creating standalone Java applications which makes use of JBindery. The old MW StandAlone method is still available for your use. But it is recommended that you use the newer method. Future version of CodeWarrior will no longer support the old method.

This chapter will walk you through using both methods to create standalone applications:

- The New JBindery Method
- The MW StandAlone Method

The New JBindery Method

Using the new method, CodeWarrior "binds" the neccessary code to your Java code to make standalone applications via an application called JBindery.

Topics discussed in this section are:

- About the JBindery Application
- Creating a Standalone Application

About the JBindery Application

JBindery is an application that binds the VM interface to Java code to create standalone Java applications. It is supplied with the Mac OS Runtime for Java (MRJ) software development kit from Apple Computer Inc. MRJ can be found on the CodeWarrior CD in the **Cool Demos, SDKs, & Tools** folder. For more information on JBindery and other parts of the MRJ SDK, see the documentation provided with MRJ.

Creating a Standalone Application

Following is a step-by-step guide to creating a standalone Mac OS Java application using JBindery.

1. **Create a new Java project.**

   Choose **New Project** from the File menu. Select `Java Application` from the Java section and click **OK**. CodeWarrior will ask you to name and save the new project to your hard disk via a standard save file dialog box.

2. **Add your Java files.**

   Choose **Add Files** from the Project menu to add your Java source files to the project. Make sure you add your files to the correct build target if you have more than one build target in your project.

3. **Open the Target Settings dialog box**

   Choose **Java Application Settings…** from the Edit menu to open the Target Settings dialog box.

   NOTE: The wording of this menu item changes to reflect the name of the build target being displayed in the project window. If you changed the name of your build target, then look for that name in this menu.

4. **Activate the Java Mac OS Post Linker.**

   Click **Target Settings** in the panel list on the left-hand side of the dialog box. The Target Settings panel appears. Choose **Java Mac OS Post Linker** from the Post-Linker popup menu This activates the Java Mac OS Post Linker and tells CodeWarrior to use the settings in the **Java MacOS Settings** panel when building your project

5. **Change the main class setting.**

   Click **Java Target** in the panel list on the left-hand side of the dialog box. The **Java Application Settings** dialog box appears. Click on **Target Settings** in the panel list to the left of the dialog box to display the Target Settings panel. Change the **Main Class** field to the same name as your application's main class.

6. **Set the Output Type.**

   Select **Java MacOS Settings** from the panel list on the left-hand side of the dialog box. This displays the Java MacOS Settings panel.

   Choose **JBindery** from the MacOS Java Output Type popup menu.

7. **Redirect stdin and stdout.**

   If you want `stdin` and `stdout` to be redirected or ignored, choose the appropriate menu items in the two **Redirect** popup menus.

8. **Set the Creator and Output Filename.**

   Change the **Creator** and **Output Filename** fields to the creator and filename you want your application to have.

9. **Set the Minimum and Maximum heap sizes.**

   Set the **Minimum app heap** to the minimum number of bytes your application will be allowed to run in. Set the **Maximum app heap** to the prefered number of bytes for your applications heap.

10. **Save your settings.**

    Click the **Save** button at the bottom of the dialog box to save all of your settings. Then close the dialog box.

11. **Compile and run.**

    Choose **Make** from the Project menu to create your application. Once it has been successfully built, double-click it's icon to run it.

The MW StandAlone Method

Just like the preferred method (JBindery), this method allows you to build standalone applications. But, instead of using JBindery, CodeWarrior combines the VM interface with your Java code to make standalone applications that use an application called MW StandAlone to intepret the Java code.

WARNING! MW StandAlone will not be supported in future versions of CodeWarrior. It is recommeded that you use the JBindery method, discussed earlier in this chapter, instead.

Topics discussed in this section are:

- MW StandAlone Features and Limitations
- Creating a Standalone Application

MW StandAlone Features and Limitations

Supported Java VMs include:

- Metrowerks Java VMs through v1.1.5
- Metrowerks JITs through v1.1.5
- Apple MRJ Library v1.0.2, 1.5, and 2.0

Class libraries can be imported weak (no project inspector options set), if you want the end-user's VM to resolve library references, or merged into output, if you want the class libraries to be wrapped into the application. Wrapping the class files into the application generates a considerably larger file.

WARNING! The VM may crash if the class libraries are not found at runtime.

If the JVM is set to 'Metrowerks' or 'Metrowerks - JIT', the file 'Metrowerks VM' & the folders 'Classes' & 'Lib' _must_ be in the same folder as the StandAlone MacOS application. If they aren't, StandAlone will look for MRJ.

Depending on which Java VM you are using, standalone applications created by MW StandAlone will look for the VM in the following places:

- **MRJ**: `System Folder:Extensions:MRJ Libraries:MRJLib` or the same folder as the generated standalone application
- **Metrowerks Java and Metrowerks JIT** : `System Folder:Extensions` or the same folder as the generated standalone

Creating a Standalone Application

Following is a step-by-step guide to creating a standalone Mac OS Java application that uses MW StandAlone.

1. **Create a new Java project.**

   Choose **New Project** from the File menu. Select `Java Application` from the Java section and click **OK**. CodeWarrior will ask you to name and save the new project to your hard disk via a standard save file dialog box.

2. **Add your Java files.**

   Choose **Add Files** from the **Project** menu to add your Java source files to the project. Make sure you add your files to the correct build target if you have more than one build target in your project.

3. **Open the Target Settings dialog box**

   Choose **Java Application Settings…** from the **Edit** menu to open the Target Settings dialog box.

NOTE: The wording of this menu item changes to reflect the name of the build target being displayed in the project window. If you changed the name of your build target, then look for that name in this menu.

---

4. **Activate the Java Mac OS Post Linker.**

   Click **Target Settings** in the panel list on the left-hand side of the dialog box. The Target Settings panel appears. Choose **Java Mac OS Post Linker** from the **Post-Linker** popup menu This activates the Java Mac OS Post Linker and tells CodeWarrior to use the settings in the **Java MacOS Settings** panel when building your project

5. **Change the main class setting.**

   Click **Java Target** in the panel list on the left-hand side of the dialog box. The **Java Application Settings** dialog box appears. Click on **Target Settings** in the panel list to the left of the dialog box to display the Target Settings panel. Change the **Main Class** field to the same name as your application's main class.

6. **Set the Output Type.**

   Select **Java MacOS Settings** from the panel list on the left-hand side of the dialog box. This displays the Java MacOS Settings panel.

   Choose **MW StandAlone** from the MacOS Java Output Type popup menu.

7. **Set the Creator and Output Filename.**

   Change the **Output Filename** and **Creator** fields to the filename and creator you want your application to have.

8. **Save your settings.**

   Click the **Save** button at the bottom of the dialog box to save all of your settings. Then close the dialog box.

9. **Compile and run.**

   Choose **Make** from the Project menu to create your application. Once it has been successfully built, double-click it's icon to run it.

Index

A

-accel option 142
accelerated interpreter 142
Add button
  Compile window 151
  Compiler window 154
  in Classpaths dialog 138, 147, 150
Add Classpaths
  Metrowerks Java 136
Add Classpaths button 144
  Native Interface window 155
Add Directory command
  Class Wrangler 121
Add File button
  Class Wrangler 115
Add Files command
  Class Wrangler 119
Add links to the Java API docs
  JavaDoc Project panel 103
All exceptions command 107
Apple MRJ VM 156
Applet
  defined 57
Applet project type
  Java Target panel 81
 tag 145
AppletFrame.java 59
applets
  choosing viewer 107
  debugging 106–110
  running 144
AppletViewer 144
  Metrowerks Java 132
  Run button 146
Application
  defined 58
  Metrowerks Java 132
Application project type
  Java Target panel 82
applications
  debugging 106–110
  options for running 139
  running 137
Assembler view
  in MWDebug 109
-asyncgc option 142

B

BCI See bytecode interpreter
Binary Transfer
  FTP Post-Linker panel 91
bytecode
  compared to object code 15
  defined 14
  verifying 143
bytecode interpreter defined 17
Bytecode Verification option
  VM preferences 159

C

C Header Style option 149
C, implementing methods in 154
Check Syntax option 153
class files
  compiling, with Sun Java compiler 151
  debugging 105–110
  disassembling 109, 146–150
  verifying 150
class folder 64, 67
Class Folder option
  Java Output panel 99
Class folder option
  Name field 99
Class Names field 148, 155
class pathnames, setting 141
Class Wranger
  Factory button 125
  Filename truncation preference 126
Class Wrangler 113–130
  Add Directory command 121
  Add File button 115
  Add Files command 119
  Compare Files command 129
  Compare Using Contents option 130
  Comparing Archives 129–130
  Compressed button 115
  Delete button 115
  Edit Manifest command 113, 124
  Extract Item button 115
  Generate Manifest button 115

Get Info button 115
JavaBean column 115
Open items collapsed preference 129
Preferences 125–??, 125, ??–129
Prompt for file at startup preference 129
Show full pathname preference 128
Use full package name for type ahead 126
Verify manifest when opening files preference 129
Class Wrangler, Type ahead selection delay preference 126
`classes.zip` file 67
`-classpath` option 141
Classpaths
Metrowerks Java 136
CodeWarrior
`Documentation` folder 11
integrated development environment (IDE) 18
tutorials 10
`CodeWarrior Documentation` folder 11
Compare Files command
Class Wrangler 129
Compare Using Contents
Class Wrangler 130
Comparing Archives
Class Wrangler 129–130
Compiler
Metrowerks Java 134
compiling, with Sun Java compiler 151
Compress
Java Output panel 100
Compressed button
Class Wrangler 115
Creator field
Java Output panel 100

D

Debug all class files in directory hierarchy (Debugger) 111
Debugger User Guide 10, 20, 105
debugging Java code 105–110
Debugging option 153
Delete button
Class Wrangler 115
Delete class files from output directory before linking
Java Output panel 99
development tools
CodeWarrior IDE (integrated development environment) 18
Disassemble option 149
Disassembler
Metrowerks Java 134, 147, 150
disassembling
class file 146–150
class files 109
documentation
`CodeWarrior Documentation` folder 11
Debugger User Guide 10, 20, 105
IDE User Guide 10, 18

E

Edit Manifest command
Class Wrangler 113, 124
encoding options
JavaDoc Project panel 103
Exceptions in targeted classes command 107
exceptions, breaking on 106
Extract Item button
Class Wrangler 115

F

Factory button
Class Wranger 125
File Type field
Java Output panel 100
Filename truncation
Class Wrangler 126
Filename Truncation menu
Java Output panel 99
Filesystem Access option
VM preferences 159
Filter using file contents
in Add Files dialog box 120
in Open dialog boxes 117
Filter using file extension
in Add Files dialog box 120
in Open dialog boxes 117
Firewall Proxy Port field

VM preferences 160
Firewall Proxy Server field
VM preferences 160
Folder to Upload
FTP Post-Linker panel 91
Force Recompile option 153
FTP Post-Linker panel 89–91
Binary Transfer 91
Folder to Upload 91
Generate Log 91
Host Address 90
Password 91
Remote Directory 91
User Name 91

G

garbage collection 142
Generate button 155
Generate Coments in Headers
Java Language panel 88
Generate Headers for All Methods
Java Language panel 88
Generate hierarchy
JavaDoc Project panel 102
Generate index
JavaDoc Project panel 102
Generate Log
FTP Post-Linker panel 91
Generate Mac-friendly filenames
JavaDoc Project panel 101
Generate Manifest
Java Output panel 100
Generate Manifest button
Class Wrangler 115
Get Info button
Class Wrangler 115

H

Header option 155
heap size, setting 141
`-help` option 143
Host Address
FTP Post-Linker panel 90
HTML file creator code
JavaDoc Project panel 102
HTTP Proxy Port field
VM preferences 159
HTTP Proxy Server field
VM preferences 159

I

IDE. *See* CodeWarrior, integrated development environment
IDE User Guide 10, 18
Ignore @author comments
JavaDoc Project panel 101
Ignore @deprecated comments
JavaDoc Project panel 101
Ignore @version comments
JavaDoc Project panel 101
installing Java 13
Interpreter 137–143
accelerated 142
Metrowerks Java 132
options 139
Interpreter window
Metrowerks Java 139

J

Jar file 59, 64, 67
and manifest information 115, 129
defined 100
Jar File option
Java Output panel 100
Jar files 58
Jar files (Java Archive files) 116
Java
bytecodes 14
installing 13
overview 14
system requirements 13
tutorial 21–55
virtual machine 14
Java Developer's Kit (JDK) 131
Java Language panel
Generate Comments in Headers 88
Generate Headers for All Methods 88
Java MacOS Post Linker Settings 91
Java MacOS Post Linker settings 91–??

Java MacOS Settings 91–??
Java Output panel 98–??
  Class folder option 99
  compress 100
  Creator field 100
  Delete class files from output directory before linking 99
  File Type field 100
  Filename Truncation menu 99
  Generate Manifest 100
  Jar File option 100
Java platform 18
Java security 158
Java Target panel 80–??
  Applet project type 81
  Application project type 82
  Library project type 84
  Main Class field 83
  Parameters field 83
  Working Directory field 84
Java Virtual Machine registers
  viewing 109
JavaBean 115
JavaBean column
  Class Wrangler 115
JavaDoc 71–??
  using 71–??
JavaDoc Project panel 101–??
  Add links to the Java API docs 103
  Encoding Options 103
  Generate hierarchy 102
  Generate Mac-friendly filenames 101
  HTML file creator code 102
  Ignore @author comments 101
  Ignore @deprecated comments 101
  Ignore @version comments 101
  Output as folder hierarchy 102
  Output encoding 103
  Scope popup menu 102
  Source file encoding 103
JavaDocProject panel
  Generate index 102
javap 146–151
JBindery 92, 163
  Creating a Standalone Application 164
JNI (Java Native Interface) 155
JNI style option
  Native Interface window 155

L

Library
  defined 58
library
  creating project 65
Library project type
  Java Target panel 84
linker option 79
Locate button
  in AppletViewer Window 145

M

Mac OS platform 18
Mac OS Runtime for Java 107
Mac OS Zip 97
Main Class field
  Java Target panel 83
memory size, for applications 141
methods, in C 154
Metro werks Java
  as bytecode interpreter 14
Metrowerks Java
  Add Classpaths 136
  AppletViewer 132
  Application 132
  Classpaths 136
  Compiler 134
  Disassembler 134, 147, 150
  Interpreter 132
  Interpreter window 139
  Native Interface 134
  Output Window 135
  VM preferences 158
Metrowerks Java application 131–160
Metrowerks JIT VM 156
Metrowerks Standard VM 156
Mixed view
  in MWDebug 109
MRJ 107
`-ms` option 141
Multi-language debugging 106

MW StandAlone
Creating a Standalone Application 167
Features and Limitations 166
MW Standalone 95
MWDebug
Assembler view 109
Mixed view 109
`-mx` option 141

N

Name field
Class folder option 99
`native` class methods 154
Native Interface
JNI style 155
Metrowerks Java 134
Native Methods 154
Network Access option
VM preferences 159
network security, for AppletViewer 146
No exceptions command 107
No Verify option 148
`-noaccel` option 142
`-noasyncgc` option 142
Normal option 153
`-noverbose` option 142
`-noverbosegc` option 142
`-noverify` option 143
`-nv` option 142

O

object code
compared to bytecode 15
Open all class files in directory hierarchy 107
Open items collapsed
Class Wrangler 129
Optimized option 153
options
for running applications 139
`-oss` option 141
Output as folder hierarchy
JavaDoc Project panel 102
output directory option 80
Output encoding
JavaDoc Project panel 103
Output Name field 155
Output Window
Metrowerks Java 135

P

Parameters field
Java Target panel 83
Password
FTP Post-Linker panel 91
platforms
Java 18
Mac OS 18
Win32 18
Post-linker option 80
PowerPC processor 18
Pre-linker option 80
processors
PowerPC 18
68K 18
x86 18
Program Preferences
Apple MRJ 156
Metrowerks Standard 156
Startup Window 157
Use VM 156
Program preferences
Metrowerks JIT 156
Use Internet Config for File Mappings 157
Use Sun AppletViewer 158
project
for library 65
Project Stationery
described 58
Prompt for file at startup
Class Wrangler 129
proxy servers 146

R

registers, for Java Virtual Machine 109
Reload VM (menu command) 136
Remote Directory
FTP Post-Linker panel 91
Run button
AppletViewer 146

running
applets 144
options for applications 139
running applications 137

S

Scope popup menu
JavaDoc Project panel 102
security permissions 158
Show all files
in Add Files dialog box 120
in Open dialog boxes 117
Show full pathname
Class Wrangler 128
Show Line Number Tables option 149
Show Line Number tables option 149
Show Private option 149
Show Processes command (MWDebug) 108
Show Registers command 109
68K processor 18
size, setting 141
Source file encoding
JavaDoc Project panel 103
`-ss` option 141
Startup Window
Program Preferences 157
`-stat` option 143
status
messages, printing 142
of javai 143
Stubs option 155
Sun Compiler 151
Sun Java compiler 151
system requirements 13

T

target name option 79
Target Settings panel 78–80
thread 141
memory size 142
Troubleshooting 161–162
tutorial 21–55
Tutorials, CodeWarrior 10
type ahead selection 116
Type ahead selection delay
Class Wrangler 126

U

Unicode 103
URL field 145
Use full package name for type ahead
Class Wrangler 126
Use Internet Config for File Mappings
Program preferences 157
Use Sun AppletViewer
Program preferences 158
Use VM
Program Preferences 156
User Name
FTP Post-Linker panel 91

V

`-v` option 142
Verbose option 149, 153
`-verbose` option 142
`-verbosegc` option 142
Verify manifest when opening files
Class Wrangler 129
Verify option 151
`-verify` option 143
verifying bytecodes 143
verifying classes 150
`-verifyremote` option 143
viewing
Java Virtual Machine registers 109
virtual machine
and platform-independent code 14
defined 14
Virtual Machine registers, Java 109
VM Preferences
Filesystem Access option 159
Firewall Proxy Server field 160
HTTP Proxy Port field 159
HTTP Proxy Server field 159
Network Access option 159
VM preferences
Bytecode Verification option 159
Firewall Proxy Port field 160

Metrowerks Java 158

W

Warnings option 153
Win32 platform 18
Working Directory field
Java Target panel 84
working directory, setting 141
-working option 141
Wrangler, Class 113–130

X

x86 processor 18

Z

zip files 58

CodeWarrior

Targeting the Java VM

From the Mac OS

Credits

writing lead:	David Blache
other writers:	L. Frank Turovich, Marc Paquette, Chris Magnuson, Derek Saldana, Jim Trudeau
engineering:	Greg Bolsinga, Kevin Buettner, John Cortell, Bernie Estavillo, Michael Farrar, Tim Freehill, Michael Stricklin
frontline warriors:	Greg Bolsinga, Barry Cameron, Pedro Larios, David Mark, Michael Stricklin, Ferry Sutanto, and CodeWarrior users everywhere

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting Java	How to use CodeWarrior to program for Java
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows